

# 第一部分 预备知识

## 第1章 C++程序设计

大家好！现在我们将要开始一个穿越“数据结构、算法和程序”这个抽象世界的特殊旅程，以解决现实生活中的许多难题。在程序开发过程中通常需要做到如下两点：一是高效地描述数据；二是设计一个好的算法，该算法最终可用程序来实现。要想高效地描述数据，必须具备数据结构领域的专门知识；而要想设计一个好的算法，则需要算法设计领域的专门知识。

在着手研究数据结构和算法设计方法之前，需要你能够熟练地运用 C++ 编程并分析程序，这些基本的技能通常是从 C++ 课程以及其他分散的课程中学到的。本书的前两章旨在帮助你回顾一下这些技能，其中的许多内容你可能已经很熟悉了。

本章我们将回顾 C++ 的一些特性。因为不是针对 C++ 新手，因此没有介绍诸如赋值语句、if 语句和循环语句（如 for 和 while）等基本结构，而是主要介绍一些可能已经被你忽略的 C++ 特性：

- 参数传递方式（如传值、引用和常量引用）。
- 函数返回方式（如返回值、引用和常量引用）。
- 模板函数。
- 递归函数。
- 常量函数。
- 内存分配和释放函数：new 与 delete。
- 异常处理结构：try, catch 和 throw。
- 类与模板类。
- 类的共享成员、保护成员和私有成员。
- 友元。
- 操作符重载。

本章中没有涉及的其他 C++ 特性将在后续章节中在需要的时候加以介绍。本章还给出了如下应用程序的代码：

- 一维和二维数组的动态分配与释放。
- 求解二次方程。
- 生成  $n$  个元素的所有排列方式。
- 寻找  $n$  个元素中的最大值。

此外，本章还给出了如何测试和调试程序的一些技巧。

### 1.1 引言

在检查程序的时候我们应该问一问：

- 它正确吗？

- 它容易读懂吗？
- 它有完善的文档吗？
- 它容易修改吗？
- 它在运行时需要多大内存？
- 它的运行时间有多长？
- 它的通用性如何？能不能不加修改就可以用它来解决更大范围的问题？
- 它可以在多种机器上编译和运行吗？或者说需要经过修改才能在不同的机器上运行吗？

上述问题的相对重要性取决于具体的应用环境。比如，如果我们正在编写一个只需运行一次即可丢弃的程序，那么主要关心的应是程序的正确性、内存需求、运行时间以及能否在一台机器上编译和运行。不管具体的应用环境是什么，正确性总是程序的一个最重要的特性。一个不正确的程序，不管它有多快、有多么好的通用性、有多么完善的文档，都是毫无意义的（除非它变正确了）。尽管我们无法详细地介绍提高程序正确性的技术，但可以为大家提供一些程序正确性的验证方法以及公认的一些良好的程序设计习惯，它们可以帮助你编写正确的代码。我们的目标是教会你如何开发正确的、精致的、高效的程序。

## 1.2 函数与参数

### 1.2.1 传值参数

考察函数 $Abc$ （见程序1-1）。该函数用来计算表达式  $a+b+b*c+(a+b-c)/(a+b)+4$ ，其中 $a$ ， $b$ 和 $c$ 是整数，结果也是一个整数。

程序1-1 计算一个整数表达式

```
int Abc(int a, int b, int c)
{
    return a+b+b*c+(a+b-c)/(a+b)+4;
}
```

在程序1-1中， $a$ ， $b$ 和 $c$ 是函数 $Abc$ 的形式参数（formal parameter），类型均为整型。如果在如下语句中调用函数 $Abc$ ：

```
z = Abc(2,x,y)
```

那么，2， $x$ 和 $y$ 分别是对应于 $a$ ， $b$ 和 $c$ 的实际参数（actual parameter）。当 $Abc(2,x,y)$ 被执行时， $a$ 被赋值为2； $b$ 被赋值为 $x$ ； $c$ 被赋值为 $y$ 。如果 $x$ 和/或 $y$ 不是int类型，那么在把它们值赋给 $b$ 和 $c$ 之前，将首先对它们进行类型转换。例如，如果 $x$ 是float类型，其值为3.8，那么 $b$ 将被赋值为3。在程序1-1中，形式参数 $a$ ， $b$ 和 $c$ 都是传值参数（value parameter）。

运行时，与传值形式参数相对应的实际参数的值将在函数执行之前被复制给形式参数，复制过程是由该形式参数所属数据类型的复制构造函数（copy constructor）完成的。如果实际参数与形式参数的数据类型不同，必须进行类型转换，从实际参数的类型转换为形式参数的类型，当然，假定这样的类型转换是允许的。

当函数运行结束时，形式参数所属数据类型的析构函数（destructor）负责释放该形式参数。当一个函数返回时，形式参数的值不会被复制到对应的实际参数中。因此，函数调用不会修改实际参数的值。

### 1.2.2 模板函数

假定我们希望编写另外一个函数来计算与程序 1-1 相同的表达式，不过这次  $a$ ， $b$  和  $c$  是 `float` 类型，结果也是 `float` 类型。程序 1-2 中给出了具体的代码。程序 1-1 和 1-2 的区别仅在于形式参数以及函数返回值的数据类型。

程序 1-2 计算一个浮点数表达式

```
float Abc(float a, float b, float c)
{
    return a+b+b*c+(a+b-c)/(a+b)+4;
}
```

实际上不必对每一种可能的形式参数的类型都重新编写一个相应的函数。可以编写一段通用的代码，将参数的数据类型作为一个变量，它的值由编译器来确定。程序 1-3 中给出了这样一段使用 `template` 语句编写的通用代码。

程序 1-3 利用模板函数计算一个表达式

```
template<class T>
T Abc(T a, T b, T c)
{
    return a+b+b*c+(a+b-c)/(a+b)+4;
}
```

利用这段通用代码，通过把 `T` 替换为 `int`，编译器可以立即构造出程序 1-1，把 `T` 替换为 `float` 又可以立即构造出程序 1-2。事实上，通过把 `T` 替换为 `double` 或 `long`，编译器又可以构造出函数 `Abc` 的双精度型版本和长整型版本。把函数 `Abc` 编写成模板函数可以让我们不必了解形式参数的数据类型。

### 1.2.3 引用参数

程序 1-3 中形式参数的用法会增加程序的运行开销。例如，我们来考察一下函数被调用以及返回时所涉及的操作。假定  $a$ ， $b$  和  $c$  是传值参数，在函数被调用时，类型 `T` 的复制构造函数把相应的实际参数分别复制到形式参数  $a$ ， $b$  和  $c$  之中，以供函数使用；而在函数返回时，类型 `T` 的析构函数会被唤醒，以便释放形式参数  $a$ ， $b$  和  $c$ 。

假定数据类型为用户自定义的 `Matrix`，那么它的复制构造函数将负责复制其所有元素，而析构函数则负责逐个释放每个元素（假定 `Matrix` 已经定义了操作符 `+`，`*` 和 `/`）。如果我们用具有 1000 个元素的 `Matrix` 作为实际参数来调用函数 `Abc`，那么复制三个实际参数给  $a$ ， $b$  和  $c$  将需要 3000 次操作。当函数 `Abc` 返回时，其析构函数又需要花费额外的 3000 次操作来释放  $a$ ， $b$  和  $c$ 。

在程序 1-4 所示的代码中， $a$ ， $b$  和  $c$  是引用参数（reference parameter）。如果用语句 `Abc(x,y,z)` 来调用函数 `Abc`，其中  $x$ 、 $y$  和  $z$  是相同的数据类型，那么这些实际参数将被分别赋予名称  $a$ ， $b$  和  $c$ ，因此，在函数 `Abc` 执行期间， $x$ 、 $y$  和  $z$  被用来替换对应的  $a$ ， $b$  和  $c$ 。与传值参数的情况不同，在函数被调用时，本程序并没有复制实际参数的值，在函数返回时也没有调用析构函数。

程序1-4 利用引用参数计算一个表达式

```
template<class T>
T Abc(T& a, T& b, T& c)
{
    return a+b+b*c+(a+b-c)/(a+b)+4;
}
```

我们可以考察一下当 $a$ ,  $b$  和 $c$  所对应的实际参数 $x$ ,  $y$  和 $z$  分别是具有1000个元素的矩阵时的情形。由于不需要把 $x$ ,  $y$  和 $z$  的值复制给对应的形式参数, 因此我们可以节省采用传值参数进行参数复制时所需要的3000次操作。

### 1.2.4 常量引用参数

C++还提供了另外一种参数传递方式——常量引用 (const reference), 这种模式指出函数不得修改引用参数。例如, 在程序 1-4中,  $a$ ,  $b$  和 $c$  的值不能被修改, 因此我们可以重写这段代码, 见程序 1-5。

程序1-5 利用常量引用参数计算一个表达式

```
template<class T>
T Abc(const T& a, const T& b, const T& c)
{
    return a+b+b*c+(a+b-c)/(a+b)+4;
}
```

使用关键字const 来指明函数不可以修改引用参数的值, 这在软件工程方面具有重要的意义。这将立即告诉用户该函数并不会修改实际参数。

对于诸如int、float 和char 的简单数据类型, 当函数不会修改实际参数值的时候我们可以采用传值参数; 对于所有其他的数据类型 (包括模板类型), 当函数不会修改实际参数值的时候可以采用常量引用参数。

采用程序 1-6的语法, 我们可以得到程序 1-5的一个更通用的版本。在新的版本中, 每个形式参数可能属于不同的数据类型, 函数返回值的类型与第一个参数的类型相同。

程序1-6 程序1-5的一个更通用的版本

```
template<class Ta, class Tb, class Tc >
Ta Abc (const Ta& a, const Tb& b, const Tc& c)
{
    return a+b+b*c+(a+b-c)/(a+b)+4;
}
```

### 1.2.5 返回值

函数可以返回值, 引用或常量引用。在前面的例子中, 函数 *Abc* 返回的都是一个具体值, 在这种情况下, 被返回的对象均被复制到调用 (或返回) 环境中。对于函数 *Abc* 的所有版本来说, 这种复制过程都是必要的, 因为函数所计算出的表达式的结果被存储在一个局部临时变量中, 当函数返回时, 这个临时变量 (以及所有其他的临时变量和传值形式参数) 所占用的空间



将被释放，其值当然也不再有效。为了避免丢失这个值，在释放临时变量以及传值形式参数的空间之前，必须把这个值从临时变量复制到调用该函数的环境中去。

如果需要返回一个引用，可以为返回类型添加一个前缀 &。如：

```
T& X(int i, T& z)
```

定义了一个函数  $X$ ，它返回一个引用参数  $z$ 。可以使用下面的语句返回  $z$ ：

```
return z ;
```

这种返回形式不会把  $z$  的值复制到返回环境中。当函数  $X$  返回时，传值形式参数  $i$  以及所有局部变量所占用的空间都将被释放。由于  $z$  是对一个实际参数的引用，因此，它不会受影响。

如果在函数名之前添加关键字 `const`，那么函数将返回一个常量引用，例如：

```
const T& X (int i, T& z)
```

除了返回的结果是一个不变化的对象之外，返回一个常量引用与返回一个引用是相同的。

### 1.2.6 递归函数

递归函数 (recursive function) 是一个自己调用自己的函数。递归函数包括两种：直接递归 (direct recursion) 和间接递归 (indirect recursion)。直接递归是指函数  $F$  的代码中直接包含了调用  $F$  的语句，而间接递归是指函数  $F$  调用了函数  $G$ ， $G$  又调用了  $H$ ，如此进行下去，直到  $F$  又被调用。在深入探讨 C++ 递归函数之前，我们来考察一下来自数学的两个相关概念——数学函数的递归定义以及归纳证明。

在数学中经常用一个函数本身来定义该函数。例如阶乘函数  $f(n)=n!$  的定义如下：

$$f(n) = \begin{cases} 1 & n \leq 1 \\ nf(n-1) & n > 1 \end{cases} \quad (1-1)$$

其中  $n$  为整数。

从该定义中可以看到，当  $n$  小于或等于 1 时， $f(n)$  的值为 1，如  $f(-3) = f(0) = f(1) = 1$ 。当  $n$  大于 1 时， $f(n)$  由递归形式来定义，在定义的右侧也出现了  $f$ 。在右侧使用  $f$  并不会导致循环定义，因为右侧  $f$  的参数小于左侧  $f$  的参数。例如，从公式 (1-1) 中可以得到  $f(2)=2f(1)$ ，由于我们已经知道  $f(1)=1$ ，因此  $f(2)=2*1=2$ 。以此类推， $f(3)=3f(2)=3*2=6$ 。

对于函数  $f(n)$  的一个递归定义 (假定是直接递归)，要想使它成为一个完整的定义，必须满足如下条件：

- 定义中必须包含一个基本部分 (base)，其中对于  $n$  的一个或多个值， $f(n)$  必须是直接定义的 (即非递归)。为简单起见，我们假定基本部分包含了  $n = k$  的情况，其中  $k$  为常数。(在基本部分中指定  $n = k$  的情形也是可能的，但较少见。)

- 在递归部分 (recursive component) 中，右侧所出现的所有  $f$  的参数都必须有一个比  $n$  小，以便重复运用递归部分来改变右侧出现的  $f$ ，直至出现  $f$  的基本部分。

在公式 (1-1) 中，基本部分是：当  $n = 1$  时  $f(n)=1$ ；递归部分是  $f(n) = nf(n-1)$ ，其中右侧  $f$  的参数为  $n-1$ ，比  $n$  要小。重复应用递归部分可把  $f(n-1)$  变换成对  $f(n-2)$ ， $f(n-3)$ ，...，直到  $f(1)$  的调用。例如：

$$f(5) = 5f(4) = 20f(3) = 60f(2) = 120f(1)$$

注意每次应用递归部分的结果是更趋近于基本部分，最后，根据基本部分的定义可以得到  $f(5) = 120$ 。从这个例子中可以看出，对于  $n = 1$  有  $f(n) = n(n-1)(n-2)\dots 1$ 。

作为递归定义的另外一个例子，我们来考察一下斐波那契数列的定义：

$$F_0=0, F_1=1, F_n=F_{n-1}+F_{n-2} \quad (n>1) \quad (1-2)$$

在这个定义中,  $F_0=0$ 和 $F_1=1$  构成了定义的基本部分,  $F_n=F_{n-1}+F_{n-2}$  是定义的递归部分。右侧函数的参数都小于  $n$ 。为使公式 (1-2) 成为  $F$  的一个完整的递归定义, 对于任何  $n>1$  的斐波那契数, 对递归部分的重复应用应能把右侧出现的所有  $F$  变换成基本部分的形式。因为对一个  $n>1$  的整数重复减去 1 或 2 会得到 0 或 1, 因此右侧  $F$  的出现总可以被变换成基本定义。比如,  $F_4=F_3+F_2=F_2+F_1+F_1+F_0=3F_1+2F_0=3$ 。

现在我们把注意力转向与计算机递归函数相关的第二个概念——归纳证明。在归纳证明中, 可以按照如下步骤来证明一个命题的正确性, 比如证明如下公式:

$$\sum_{i=1}^n i = n(n+1)/2 \quad (n \geq 0) \quad (1-3)$$

首先我们可以验证, 对于  $n$  的一个或多个基本的值 (如  $n=0$  或  $n=0,1$ ) 该公式是成立的; 然后假定当  $n$  ( $0 \sim m$ ) 时公式是成立的, 其中  $m$  是一个任意整数 (大于等于验证时所取的最大值), 如果能够证明对于  $n$  的下一个值 (如  $m+1$ ) 公式也是成立的, 那么就可以确定该公式是成立的。这种证明方法可以归纳为三个部分——归纳初值 (induction base), 归纳假设 (induction hypothesis) 和归纳步证明 (induction step)。

下面通过对  $n$  进行归纳来证明公式 (1-3)。在归纳初值部分, 取  $n=0$  来进行验证, 由于公式的左边  $\sum_{i=1}^0 i=0$ , 公式的右边也为 0, 所以当  $n=0$  时公式 (1-3) 是成立的。在归纳假设部分假定当  $n \leq m$  时公式是成立的, 其中  $m$  是任意大于等于 0 的整数 (对于接下来的归纳证明, 只需假定  $n=m$  时公式是成立的即可)。在归纳证明中需要证明当  $n=m+1$  时公式 (1-3) 是成立的。对于  $n=m+1$ , 公式左边为  $\sum_{i=1}^{m+1} i = m+1 + \sum_{i=1}^m i$ , 从归纳假设中可以知道  $\sum_{i=1}^m i = m(m+1)/2$ , 所以当  $n=m+1$  时左边变成  $m+1 + m(m+1)/2 = (m+1)(m+2)/2$ , 正好与公式右边相等。

乍看起来, 归纳证明好象是一个循环证明——因为我们建立了一个假设为正确的结论。不过, 归纳证明并不是循环证明, 就像递归定义并不是循环定义一样。每个正确的归纳证明都会有一个基本值验证部分, 它与递归定义的基本部分相类似, 在归纳证明时我们利用了比  $n$  值小时结论的正确性来证明取值为  $n$  时结论的正确性。重复应用归纳证明, 可以减少对基本值验证的应用。

C++ 允许我们编写递归函数。一个正确的递归函数必须包含一个基本部分。函数中递归调用部分所使用的参数值应比函数的参数值要小, 以便函数的重复调用能最终获得基本部分所提供的值。

例1-1 [阶乘] 程序1-7给出了一个利用公式 (1-1) 计算  $n!$  的 C++ 函数。函数的基本部分包含了  $n=1$  的情形。考虑调用 Factorial(2)。为了计算 else 语句中的  $2 * \text{Factorial}(1)$ , 需要挂起 Factorial(2), 然后进入调用 Factorial(1)。当 Factorial(2) 被挂起时, 程序的状态 (如局部变量、传值形式参数的值、引用形式参数的值以及代码的执行位置等) 被保留在递归栈中, 在执行完 Factorial(1) 时这些程序状态又立即恢复。调用 Factorial(1) 所得到的返回值为 1, 之后, Factorial(2) 恢复运行, 计算表达式  $2 * 1$ , 并将结果返回。

程序1-7 计算  $n!$  的递归函数

```
int Factorial (int n)
{//计算  $n!$ 
```

```

    if (n<=1) return 1;
    else return n * Factorial(n-1);
}

```

在计算Factorial(3)时，当到达else语句时，计算过程被挂起以便先计算出Factorial(2)。我们已经看到Factorial(2)是怎样获得最终结果2的。当Factorial(2)返回时，Factorial(3)继续运行，计算出最后的结果3\*2。

鉴于程序1-7的代码与公式(1-1)的相似性，该程序的正确性与公式(1-1)的正确性是等价的。

例1-2 模板函数Sum(见程序1-8)统计元素a[0]至a[n-1]的和(简记为a[0:n-1])。从代码中我们可以得到这样的递归公式：当n=0时，和为0；当n>0时，n个元素的和是前面n-1个元素的和加上最后一个元素。见程序1-9。

程序1-8 累加a[0 : n-1]

```

template<class T>
T Sum(T a[], int n)
{//计算a[0: n-1]的和
    T tsum=0;
    for (int i = 0; i < n; i++)
        tsum += a[i];
    return tsum;
}

```

程序1-9 递归计算a[0 : n-1]

```

template<class T>
T Rsum(T a[], int n)
{//计算a[0: n-1]的和
    if (n > 0)
        return Rsum(a, n-1) + a[n-1];
    return 0;
}

```

例1-3 [排列] 通常我们希望检查n个不同元素的所有排列方式以确定一个最佳的排列。比如，a, b 和c 的排列方式有：abc, acb, bac, bca, cab 和cba。n个元素的排列方式共有n!种。

由于采用非递归的C++函数来输出n个元素的所有排列方式很困难，所以可以开发一个递归函数来实现。令 $E=\{e_1, \dots, e_n\}$ 表示n个元素的集合，我们的目标是生成该集合的所有排列方式。令 $E_i$ 为E中移去元素 $i$ 以后所获得的集合， $perm(X)$ 表示集合X中元素的排列方式， $e_i.perm(X)$ 表示在 $perm(X)$ 中的每个排列方式的前面均加上 $e_i$ 以后所得到的排列方式。例如，如果 $E=\{a, b, c\}$ ，那么 $E_1=\{b, c\}$ ， $perm(E_1)=(bc, cb)$ ， $e_1.perm(E_1)=(abc, acb)$ 。

对于递归的基本部分，采用 $n=1$ 。当只有一个元素时，只可能产生一种排列方式，所以 $perm(E)=(e)$ ，其中e是E中的唯一元素。当 $n>1$ 时， $perm(E)=e_1.perm(E_1)+e_2.perm(E_2)+e_3.perm(E_3)+\dots+e_n.perm(E_n)$ 。这种递归定义形式是采用n个 $perm(X)$ 来定义 $perm(E)$ ，其中每个X包含n-1个元素。至此，一个完整的递归定义所需要的基本部分和递归部分都已完成。

当 $n=3$ 并且 $E=(a, b, c)$ 时, 按照前面的递归定义可得 $perm(E)=a.perm(\{b, c\})+b.perm(\{a, c\})+c.perm(\{a, b\})$ 。同样, 按照递归定义有 $perm(\{b, c\})=b.perm(\{c\})+c.perm(\{b\})$ , 所以 $a.perm(\{b, c\})=ab.perm(\{c\})+ac.perm(\{b\})=ab.c+ac.b=(abc, acb)$ 。同理可得 $b.perm(\{a, c\})=ba.perm(\{c\})+bc.perm(\{a\})=ba.c+bc.a=(bac, bca)$ ,  $c.perm(\{a, b\})=ca.perm(\{b\})+cb.perm(\{a\})=ca.b+cb.a=(cab, cba)$ 。所以 $perm(E)=(abc, acb, bac, bca, cab, cba)$ 。

注意 $a.perm(\{b, c\})$ 实际上包含两个排列方式:  $abc$  和  $acb$ ,  $a$  是它们的前缀,  $perm(\{b, c\})$  是它们的后缀。同样地,  $ac.perm(\{b\})$  表示前缀为  $ac$ 、后缀为  $perm(\{b\})$  的排列方式。

程序1-10把上述 $perm(E)$ 的递归定义转变成一个C++函数, 这段代码输出所有前缀为 $list[0:k-1]$ , 后缀为 $list[k:m]$ 的排列方式。调用 $Perm(list, 0, n-1)$ 将得到 $list[0:n-1]$ 的所有 $n!$ 个排列方式, 在该调用中,  $k=0$ ,  $m=n-1$ , 因此排列方式的前缀为空, 后缀为 $list[0:n-1]$ 产生的所有排列方式。当 $k=m$ 时, 仅有一个后缀 $list[m]$ , 因此 $list[0:m]$ 即是所要产生的输出。当 $k<m$ 时, 先用 $list[k]$ 与 $list[k:m]$ 中的每个元素进行交换, 然后产生 $list[k+1:m]$ 的所有排列方式, 并用它作为 $list[0:k]$ 的后缀。Swap是一个inline函数, 它被用来交换两个变量的值, 其定义见程序1-11。Perm的正确性可用归纳法来证明。

程序1-10 使用递归函数生成排列

```
template<class T>
void Perm(T list[], int k, int m)
//生成list[k:m]的所有排列方式
{
    int i;
    if (k == m) //输出一个排列方式
        for (i = 0; i <= m; i++)
            cout << list[i];
        cout << endl;
    }
    else // list[k:m]有多个排列方式
        // 递归地产生这些排列方式
        for (i=k; i <= m; i++) {
            Swap(list[k], list[i]);
            Perm(list, k+1, m);
            Swap(list[k], list[i]);
        }
}
```

程序1-11 交换两个值

```
template <class T>
inline void Swap(T& a, T& b)
// 交换a和b
{
    T temp = a; a = b; b = temp;
}
```

## 练习

1. 试编写一个模板函数Input, 它要求用户输入一个非负数, 并负责验证用户所输入的数是

否真的大于或等于0,如果不是,它将告诉用户该输入非法,需要重新输入一个数。在函数非成功退出之前,应给用户三次机会。如果输入成功,函数应当把所输入的数作为引用参数返回。输入成功时,函数应返回true,否则返回false。上机测试该函数。

2. 试编写一个模板函数,用来测试数组a中的元素是否按升序排列(即 $a[i] \leq a[i+1]$ ,其中 $0 \leq i < n-1$ )。如果不是,函数应返回false,否则应返回true。上机测试该函数。

3. 试编写一个非递归函数来计算 $n!$ ,并上机测试函数的正确性。

4. 1) 试编写一个计算斐波那契数列 $F_n$ 的递归函数,并上机测试其正确性。

2) 试说明对于任何 $n > 2$ 的整数,调用1)中的函数计算 $F_n$ 时,同一个 $F_i$ 会被处理至少一次。

3) 试编写一个非递归的函数来计算斐波那契数列 $F_n$ ,该函数应能直接计算出每个斐波那契数。上机测试代码的正确性。

5. 试编写一个递归函数,用来输出 $n$ 个元素的所有子集。例如,三个元素 $\{a, b, c\}$ 的所有子集是: $\{\}$ (空集), $\{a\}$ , $\{b\}$ , $\{c\}$ , $\{a, b\}$ , $\{a, c\}$ , $\{b, c\}$ 和 $\{a, b, c\}$ 。

6. 试编写一个递归函数来确定元素 $x$ 是否属于数组 $a[0:n-1]$ 。

## 1.3 动态存储分配

### 1.3.1 操作符new

C++操作符new可用来进行动态存储分配,该操作符返回一个指向所分配空间的指针。例如,为了给一个整数动态分配存储空间,可以使用下面的语句来说明一个整型指针变量:

```
int *y;
```

当程序需要使用该整数时,可以使用如下语法来为它分配存储空间:

```
y = new int;
```

操作符new分配了一块能存储一个整数的空间,并将指向该空间的指针返回给 $y$ , $y$ 是对整数指针的引用,而 $*y$ 则是对整数本身的引用。为了在刚分配的存储空间中存储一个整数值,比如10,可以使用如下语法:

```
*y = 10;
```

我们可以把上述三步(说明 $y$ ,分配存储空间,为 $*y$ 赋值)进行适当的合并,如下例所示:

```
int *y = new int;
```

```
*y = 10;
```

或

```
int *y = new int (10);
```

或

```
int *y;
```

```
y = new int (10);
```

### 1.3.2 一维数组

在本书的许多示例程序中都使用了一维或二维数组,这些数组的大小在编译时可能是未知的,事实上,它们可能随着函数调用的变化而变化。因此,对于这些数组必须进行动态存储分配。

为了在运行时创建一个一维浮点数组 $x$ ,首先必须把 $x$ 说明成一个指向float的指针,然后为数组分配足够的空间。例如,一个大小为 $n$ 的一维浮点数组可以按如下方式来创建:

```
float *x = new float [n];
```

操作符 `new` 分配 `n` 个浮点数所需要的空间，并返回指向第一个浮点数的指针。可以使用如下语法来访问每个数组元素：`x[0]`, `x[1]`, ..., `x[n-1]`。

### 1.3.3 异常处理

在执行语句

```
float *x = new float [n];
```

时，如果计算机不能分配足够的空间怎么办？在这种情况下，`new` 不能够分配所需数量的存储空间，将会引发一个异常（exception）。在 Borland C++ 中，当 `new` 不能分配足够的空间时，它会引发（throw）一个异常 `xalloc`（在 `except.h` 中定义）。可以采用 `try - catch` 结构来捕获（catch）`new` 所引发的异常：

```
float *x;
try {x = new float [n];}
catch (xalloc) { // 仅当new 失败时才会进入
    cerr << "Out of Memory" << endl;
    exit(1);}
```

当一个异常出现时，程序进入与该异常相对应的 `catch` 语句块中执行。在上面的例子中，只有在执行 `try` 语句时产生了 `xalloc` 异常，才会进入 `catch (xalloc)` 语句块，由语句 `exit (1)` 终止程序的运行。（`exit ()` 在 `stdlib.h` 中定义。）

在 C++ 程序中处理错误条件的常用方法就是每当检测到这样的条件时就引发一个异常。当一个异常被引发时，必须指明它的类型（如前面的 `xalloc`）。我们把可能会产生异常的程序代码放入 `try` 块中，在 `try` 块之后放上一个或多个 `catch` 块，每个 `catch` 块用来处理一个特定类型的异常。例如 `catch (xalloc)` 块仅处理 `xalloc` 异常。语法 `catch (...)` 定义了一个能捕获所有异常的 `catch` 块。当一个异常被引发时，检查在程序执行过程中所遇到的最接近的 `try-catch` 代码，如果其中的一个 `catch` 块能够处理所产生的异常，程序将从这个 `catch` 块中继续执行，否则，继续检查直到找到与异常相匹配的块。如果找不到能处理该异常的 `catch` 块，程序将显示信息 “Abnormal program termination（异常程序终结）” 并终止运行。如果一个 `try` 块没有引发异常，程序的执行将越过与该 `try` 块相对应的 `catch` 块。

### 1.3.4 操作符 delete

动态分配的存储空间不再需要时应该被释放，所释放的空间可重新用来动态创建新的结构。可以使用 C++ 操作符 `delete` 来释放由操作符 `new` 所分配的空间。下面的语句可以释放分配给 `*y` 的空间以及一维数组 `x`：

```
delete y;
delete [] x;
```

### 1.3.5 二维数组

虽然 C++ 提供了多种机制用来说明二维数组，但其中的多数机制都要求在编译时明确地知道每一维的大小。而且，在使用这些机制时，很难编写出一个允许形式参数是一个第二维大小未知的二维数组的函数。之所以如此，是因为当形式参数是一个二维数组时，必须指定其第二维的大小。例如，`a[ ][10]` 是一个合法的形式参数，而 `a[ ][ ]` 不是。

克服这种限制的一条有效途径就是对于所有的二维数组使用动态存储分配。本书从头至尾



使用的都是动态分配的二维数组。

当一个二维数组每一维的大小在编译时都是已知时，可以采用类似于创建一维数组的语法来创建二维数组。例如，一个类型为 `char` 的  $7 \times 5$  数组可用如下语法来定义：

```
char c[7][5];
```

如果在编译时至少有一维是未知的，必须在运行时使用操作符 `new` 来创建该数组。一个二维字符型数组，假定在编译时已知其列数为 5，可采用如下语法来分配存储空间：

```
char (*c)[5];
try { c = new char [n][5];}
catch (xalloc) {//仅当new失败时才会进入
    cerr << "Out of Memory" << endl;
    exit (1);}
```

在运行时，数组的行数 `n` 要么通过计算来确定，要么由用户来指定。如果在编译时数组的列数也是未知的，那么不可能调用一次 `new` 就能创建该数组（即使数组的行数是已知的）。构造二维数组时，可以把它看成是由若干行组合起来的，每一行都是一个一维数组，可以按照前面讨论的方式用 `new` 来创建，指向每一行的指针可以保存在另外一个一维数组之中。图 1-1 给出了建立一个  $3 \times 5$  数组所需要的结构。

`x[0]`, `x[1]`, `x[2]` 分别指向第 0 行，第 1 行和第 2 行的第一个元素。所以，如果 `x` 是一个字符数组，那么 `x[0:2]` 是指向字符的指针，而 `x` 本身是一个指向指针的指针。可用如下语法来说明 `x`：

```
char **x;
```

为了创建如图 1-1 所示的存储结构，可以使用程序 1-12 中的代码，该程序创建一个类型为 `T` 的二维数组，这个数组有 `rows` 行和 `cols` 列。程序首先为指针 `x[0], ..., x[rows-1]` 申请空间，然后为数组的每一行申请空间。在程序中操作符 `new` 被调用了 `rows+1` 次。如果 `new` 的某一次调用引发了一个异常，程序控制将转移到 `catch` 块中，并返回 `false`。如果没有出现异常，数组将被成功创建，函数 `Make2DArray` 返回 `true`。对于所创建的数组 `x` 中的元素，可以使用标准的用法来引用，如 `x[i][j]`，其中  $0 \leq i < \text{rows}, 0 \leq j < \text{cols}$ 。

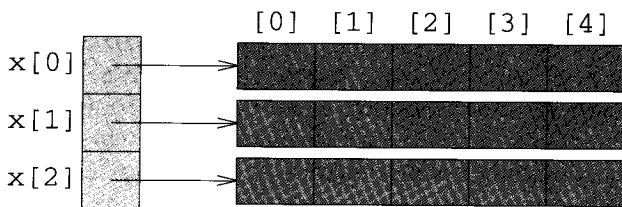


图1-1 一个  $3 \times 5$  数组的存储结构

程序1-12 为一个二维数组分配存储空间

```
template <class T>
bool Make2DArray ( T ** &x, int rows, int cols)
// 创建一个二维数组
{
    try{
        //创建行指针
        x = new T * [rows];

        //为每一行分配空间
```



```
for (int i = 0 ; i < rows; i++)
    x[i] = new int [cols];
return true;
}
catch (xalloc) {return false;}
}
```

在程序1-12中，函数通过返回布尔值false 把new所产生的异常（如果有的话）告诉调用者。当然，Make2DArray失败时也可以什么都不做，这样也能使调用者知道产生了异常。如果使用程序1-13中的代码，调用者可以捕获由new所产生的任何异常。

程序1-13 创建一个二维数组但不处理异常

```
template <class T>
void Make2DArray( T ** &x, int rows, int cols)
{
    // 创建一个二维数组
    // 不捕获异常

    //创建行指针
    x = new T * [rows];

    //为每一行分配空间
    for (int i = 0 ; i<rows; i++)
        x[i] = new int [cols];
}
```

当Make2DArray按程序1-13定义时，可以使用如下代码来确定存储分配是否成功：

```
try { Make2DArray (x, r, c);}
catch (xalloc) {cerr<< "Could bot create x" << endl;
                exit(1);}
```

在Make2DArray中不捕获异常不仅简化了函数的代码设计，而且可以使用户在一个更合适的地方捕获异常，以便更好地报告出错误的明确含义或进行错误恢复。

可以按如下两步来释放程序1-12中为二维数组所分配的空间。首先释放在for循环中为每一行所分配的空间，然后释放为行指针所分配的空间，具体实现见程序1-14。注意在程序1-14中x被置为0，以便阻止用户继续访问已被释放的空间。

程序1-14 释放由Make2DArray所分配的空间

```
template <class T>
void Delete2DArray( T ** &x, int rows)
{
    // 删除二维数组 x

    //释放为每一行所分配的空间
    for (int i = 0 ; i < rows ; i++)

        delete [ ] x[i];
    //删除行指针
    delete [] x;
    x = 0;
}
```

## 练习

7. 假定用一维数组 `a[0 : size-1]` 来存储一组元素。如果有 `n` 个元素，可以把它们存储在 `a[0], ..., a[n-1]` 中。当 `n` 超过 `size` 时，数组将不足以存储所有元素，必须分配一个更大的数组。类似地，如果元素的数目比 `size` 小很多，我们又可能希望减少数组的大小，以便释放出多余的空间为其他地方所用。试编写一个模板函数 `ChangeSize1D` 把数组 `a` 的大小从 `size` 变成 `ToSize`。函数首先应该分配一个新的、大小为 `ToSize` 的数组，然后把原数组 `a` 中的 `n` 个元素复制到新数组 `a` 中，最后释放原数组 `a` 所占用的空间。上机测试该函数。

8. 试编写一个函数 `ChangeSize2D` 来改变一个二维数组的大小（见练习7）。上机测试该函数。

## 1.4 类

## 1.4.1 类Currency

C++ 语言支持诸如 `int`, `float` 和 `char` 之类的数据类型，在本书所提供的许多应用中还使用了 C++ 语言不直接支持的数据类型。用 C++ 来定义自有数据类型最灵活的方式就是使用类（`class`）结构。假定你想处理类型 `Currency` 的对象，其实例拥有三个成员：符号（+ 或 -），美元和美分。举两个例子，如 \$2.35（符号是 +，2 美元，35 美分）和 -\$6.05（符号是 -，6 美元，5 美分）。对这种类型的对象我们想要执行的操作如下：

- 1) 设置成员的值。
- 2) 确定各成员的值（如指出符号，美元数目和美分数目）。
- 3) 增加两种货币类型。
- 4) 增加成员的值。
- 5) 输出。

假定用无符号长整型变量 `dollars`、无符号整型变量 `cents` 和 `sign` 类型的变量 `sgn` 来描述货币对象，其中 `sign` 类型的定义如下：

```
enum sign { plus, minus};
```

可以使用程序 1-15 中的语法来定义 C++ 类 `Currency`。第一行简单地说明一个名为 `Currency` 的类，然后在一对括号（`{}`）之间给出类描述。类描述被分成两个部分：`public` 和 `private`。`public` 部分用于定义一些函数（又称方法），这些函数可对 `Currency` 类对象（或实例）进行操作，它们对于 `Currency` 类的用户是可见的，是用户与 `Currency` 对象进行交互的唯一手段。`private` 部分用于定义函数和数据成员（如简单变量，数组及其他可赋值的结构），这些函数和数据成员对于用户来说是不可见的。借助于 `public` 部分和 `private` 部分，我们可以使用户只看到他（或她）需要看到的部分，同时把其余信息隐藏起来。尽管 C++ 语法允许在 `public` 部分定义数据成员，但在软件工程实践中不鼓励这种做法。

程序 1-15 定义 `Currency` 类

```
class Currency {
public:
    // 构造函数
    Currency(sign s = plus, unsigned long d = 0, unsigned int c = 0);

    // 析构函数
```

```

~Currency() {}
bool Set(sign s, unsigned long d, unsigned int c);
bool Set(float a);
sign Sign() const {return sgn;}
unsigned long Dollars() const {return dollars;}
unsigned int Cents() const {return cents;}
Currency Add(const Currency& x) const;
Currency& Increment(const Currency& x);
void Output() const;
private:
    sign sgn;
    unsigned long dollars;
    unsigned int cents;
};

```

public部分的第一个函数与Currency类同名，这种函数称之为构造函数。构造函数指明如何创建一个给定类型的对象，它不可以有返回值。在本例中，构造函数有三个参数，其缺省值分别是plus, 0和0，构造函数的具体实现在本节稍后给出。在创建一个Currency类对象时，构造函数被自动唤醒。可以采用如下两种方式来创建Currency类对象：

```

Currency f, g (plus, 3,45), h (minus, 10);
Currency *m = new Currency ( plus, 8, 12);

```

第一行定义了三个Currency类变量（f，g 和h），其中f 被初始化为缺省值plus, 0和0，而g被初始化为\$3.45，h 被初始化为 - \$10.00。注意初始值从左至右分别对应构造函数的每个参数。如果初始值的个数少于构造函数参数的个数，剩下的参数将取缺省值。在第二行，m被定义为指向一个Currency对象的指针。我们调用new函数来创建一个Currency对象，并把对象的指针存储在m中。所创建的对象被初始化为\$8.12。

下一个函数为~Currency，与类名相比多了一个前缀（~），这个函数被称为析构函数。每当一个Currency对象超出作用域时将自动调用析构函数。这个函数用来删除对象。在本例中析构函数被定义为空函数（{}）。对于其他类，由于类的构造函数可能会创建一些动态数组，那么当对象超出作用域时，析构函数需要释放这些空间。与构造函数一样，析构函数也不可以有返回值。

接下来的两个函数允许用户为Currency类成员赋值。其中第一个函数要求用户提供三个参数，而第二个函数需要一个浮点数作为参数。如果成功，两个函数均返回 true，否则返回false。这两个函数的具体实现在本节稍后给出。请注意，这两个函数具有相同的名字，但编译器和用户都很容易区分它们，因为它们具有不同的参数集合。C++允许函数名的重用，只要它们的参数表不同。还需要注意的是，没有指定欲赋值（符号，美元，美分）对象的名称，这是因为调用类成员函数的语法如下：

```

g.Set(minus,33,0);
h.Set(20.52);

```

其中g 和h 是Currency 类变量。在第一个句子中，g 是唤醒Set 的对象，而在第二个句子中h是唤醒Set 的对象。在为函数Set编写代码时，我们有办法访问调用本函数的对象，因此，不需要把对象的名称放入参数表中。

函数Sign，Dollars和Cents返回对象的相应数据成员，关键字const指出这些函数不会修改数据成员。我们把这种类型的函数称之为常元函数（constant function）。

函数Sum把当前对象的货币数量与对象x的货币数量相加，然后返回所得结果，因此Add函数不会修改当前对象，是一个常元函数。函数Increment把对象x的货币数量添加到当前对象上，这个函数修改了当前对象，因此不是一个常元函数。最后一个函数是Output，它显示当前对象的货币数量。函数Output不会修改当前对象，因此是一个常元函数。

尽管Add和Increment都返回Currency类对象，但Add返回的是值，而Increment返回的是引用。如1.2.5节所提到的，返回值和返回引用分别与传值参数和引用参数有相同的作用。在返回一个值的情况下，返回的对象被复制到所返回的环境，而返回引用则避免了这种复制，在返回的环境中可以直接使用该对象。返回引用比返回值要快，因为省去了复制过程。从Add的代码中可以看出，它返回了一个局部对象，在函数终止时该对象将被删除，因此，return语句必须复制该对象。而Increment返回的是一个全局对象，因而不需要复制。

复制构造函数被用来执行返回值的复制及传值参数的复制。程序1-15中没有给出复制构造函数，所以C++将使用缺省的复制构造函数，它仅可进行数据成员的复制。对于类Currency来说，使用省缺的复制构造函数已经足够。后面还将看到许多类，对于这些类缺省的复制构造函数已难以胜任它们的复制工作。

在private部分，定义了三个数据成员，它们对于一个Currency对象来说是必须的。每一个Currency对象都拥有自己的这三个数据成员。

由于在类定义的内部没有给出函数的具体实现，因此必须在其他地方给出。在具体实现时，必须在每个函数名的前面加上Currency::，以指明该函数是Currency类的成员函数。所以Currency::Currency表示该函数是Currency类的构造函数，而Currency::Output表示该函数是Currency类的Output函数。程序1-16给出了Currency类的构造函数。

程序1-16 Currency类的构造函数

```
Currency::Currency(sign s, unsigned long d, unsigned int c)
{
    // 创建一个Currency对象
    if(c > 99)
    {
        // 美分数目过多
        cerr << "Cents should be < 100" << endl;
        exit(1);
    }

    sgn = s; dollars = d; cents = c;
}
```

构造函数在初始化当前对象的sgn, dollars和cents数据成员之前需要验证参数的合法性。如果参数值出现错误，构造函数将输出一个错误信息，然后调用函数exit()终止程序的运行。在本例中，仅需要验证c的值。

程序1-17给出了两个Set函数的代码。第一个函数首先验证参数的合法性，如果参数合法，则用它们来设置private成员变量。第二个函数不执行参数合法性验证，它仅使用小数点后面的头两个数字。形如 $d_1.d_2d_3$ 的数可能没有一个精确的计算机表示，例如，用计算机所描述的数5.29实际上要比真正的5.29稍微小一点。当用如下语句

```
cents = (a - dollars) * 100
```

抽取cents成员时，这种描述方法可能会带来一个错误，因为 $(a - dollars) * 100$ 稍微小于29，当程序把 $(a - dollars) * 100$ 转换成整数时，cents得到的将是28而不是29。只要 $d_1.d_2d_3$ 的计算机表示与实际值相比不少于0.001或不多于0.009，就可以采用为a加上0.001来解决我们的问

题。例如，如果5.29的计算机表示是5.28999，那么加上0.001将得到5.29099,由此所计算出的cents就是29。

程序1-17 设置private数据成员

---

```
bool Currency::Set(sign s, unsigned long d, unsigned int c)
{
    // 取值
    if (c > 99) return false;
    sgn = s; dollars = d; cents = c;
    return true;
}

bool Currency::Set(float a)
{
    // 取值
    if (a < 0) {sgn = minus; a = -a;}
    else sgn = plus;
    dollars = a; // 抽取整数部分
    // 获取两个小数位
    cents = (a + 0.005 - dollars) * 100;
    return true;
}
```

---

程序1-18给出了函数Add的代码，该函数首先把要累加的两个货币数量转换成整数，如\$2.32 变成整数232，-\$4.75变成整数-475。请注意引用当前对象的数据成员与引用参数x的数据成员在语法上有所区别。x.dollars指定x的数据成员dollars，而当前对象使用dollars时可以直接引用dollars而不必在它的前面加上对象名。当函数Add终止时，局部变量a1,a2,a3和ans被long数据类型的析构函数删除，这些变量所占用的空间也将被释放。由于Currency对象ans将被作为调用结果返回，因此必须把它复制到调用者的环境中，所以Add返回的是值。

程序1-18 累加两个Currency

---

```
Currency Currency::Add(const Currency& x) const
{
    // 把x累加到*this.
    long a1, a2, a3;
    Currency ans;
    // 把当前对象转换成带符号的整数
    a1 = dollars * 100 + cents;
    if (sgn == minus) a1 = -a1;

    // 把x转换成带符号的整数
    a2 = x.dollars * 100 + x.cents;
    if (x.sgn == minus) a2 = -a2;

    a3 = a1 + a2;

    // 转换成 currency 形式
    if (a3 < 0) {ans.sgn = minus; a3 = -a3;}
    else ans.sgn = plus;
    ans.dollars = a3 / 100;
}
```

---

```
ans.cents = a3 - ans.dollars * 100;

return ans;
}
```

程序1-19给出了函数Increment和Output的代码。在C++中，保留关键字this用于指向当前对象，\*this 代表对象本身。看一下调用g.Increment(h)。函数Increment的第一行调用了public成员函数Add，它把x(这里是h) 加到当前对象上( 这里是 g )，所得结果被返回，并被赋给 \*this，\*this就是当前对象。由于该对象不是函数Increment的局部对象，因此当函数结束时，该对象不会自动被删除。所以可以返回一个引用。

程序1-19 Increment与Output

```
Currency& Currency::Increment(const Currency& x)
{// 增加量 x.
    *this = Add(x);
    return *this;
}

void Currency::Output () const
{// 输出currency 的值
    if (sgn == minus) cout << '-';
    cout << '$' << dollars << '.';
    if (cents < 10) cout << "0";
    cout << cents;
}
```

通过把Currency类的成员变成私有( private )，我们可以拒绝用户访问这些成员，所以用户不能使用如下的语句来改变这些成员的值：

```
h.cents = 20;
h.dollars = 100;
h.sgn = plus;
```

利用成员函数来设置数据成员的值可以确保数据成员拥有合法的值。构造函数和 Set函数已经做到了这一点，其他函数当然也应该保证数据成员的合法性。因此，在诸如 Add和Output函数的代码中不必验证cents是否介于0到100之间。如果数据成员被声明为public成员，它们的合法性将难以保证。用户可能会错误地把 cents设置成305，因而将导致一些函数( 如 Output函数 )产生错误结果，所以，所有的函数在处理任务之前都必须验证数据的合法性。这种验证将会降低代码的执行速度，同时也使代码不够优雅。

程序1-20给出了类Currency的应用示例。这段代码假定类定义及实现都在文件 curr1.h之中。我们一般把类定义和类实现分放在不同的文件中，然而，这种分开放置的方法可能会对后续章节中大量使用模板函数和模板类带来困难。

函数main的第一行定义了四个Currency类变量：g, h, i 和j。除h 具有初值\$3.50外，构造函数把它们都初始化为\$0.00。在接下来的两行中，g 和i 分别被设置成 - \$2.25和 - \$6.45，之后调用函数Add把g 和h 加在一起，并把所返回的对象( 值为\$1.25 )赋给j。为此，需使用缺省的赋值过程把右侧对象的各数据成员分别复制到左侧对象相应的数据成员之中，复制的结果是使 j 具有值\$1.25，这个值在下一行被输出。

下两行语句把i累加到h上,并输出i的新值-\$2.95。接下来的一行首先把i和g加在一起,然后返回一个临时对象(其值为-\$5.20),此后,把h加到这个临时对象上并返回一个新的临时对象,其值为-\$1.70。新的临时对象被复制到j中,然后输出j的值(为-\$1.70)。注意‘.’序列的处理顺序是从左到右。

接下来的一行语句首先使用Increment为i累加g,它返回一个引用给i。Add把i和h的和返回给j,最后输出j的结果为-\$1.70,i的结果为-\$5.20。

程序1-20 Currency类应用示例

```
#include <iostream.h>
#include "curr1.h"

void main (void)
{
    Currency g, h(plus, 3, 50), i, j;
    g.Set(minus, 2, 25);
    i.Set(-6.45);
    j = h.Add(g);
    j.Output(); cout << endl;
    i.Increment(h);
    i.Output(); cout << endl;
    j = i.Add(g).Add(h);
    j.Output(); cout << endl;
    j = i.Increment(g).Add(h);
    j.Output(); cout << endl;
    i.Output(); cout << endl;
}
```

#### 1.4.2 使用不同的描述方法

假定已经有许多应用采用了程序1-15中所定义的Currency类,现在我们想要对Currency类的描述进行修改,使其应用频率最高的两个函数Add和Increment可以运行得更快,从而提高应用程序的执行速度。由于用户仅能通过public部分所提供的接口与Currency类进行交互,因此对private部分的修改并不会影响应用代码的正确性。所以可以修改private部分而不会使应用发生变化。

在Currency对象新的描述中,仅有一个私有数据成员,其类型为long。数132代表\$1.32,而-20代表-\$0.20。程序1-21、1-22、1-23中给出了Currency类的新的描述方法以及各成员函数的具体实现。

注意,如果把新代码放在文件curr1.h中,则可以运行程序1-20中的代码而不需要做任何修改。对用户隐藏实现细节的一个重大好处在于可以用新的、更高效的描述来取代以前的描述而不需要改变应用代码。

程序1-21 Currency类的新定义

```
class Currency {
public:
    // 构造函数
```



```

Currency(sign s = plus, unsigned long d = 0, unsigned int c = 0);
// 析构函数
~Currency() {}
bool Set(sign s, unsigned long d, unsigned int c);
bool Set(float a);
sign Sign() const
{if (amount < 0) return minus;
 else return plus;}
unsigned long Dollars() const
{if (amount < 0) return (-amount) / 100;
 else return amount / 100;}
unsigned int Cents() const
{if (amount < 0)
    return -amount - Dollars() * 100;
 else return amount - Dollars() * 100;}
Currency Add(const Currency& x) const;
Currency& Increment(const Currency& x)
{amount += x.amount; return *this;}
void Output() const;
private:
    long amount;
};

```

程序1-22 新的构造函数及Set函数

```

Currency::Currency(sign s, unsigned long d, unsigned int c)
{// 创建Currency 对象
    if (c > 99)
        {// 美分数目过多
            cerr << "Cents should be < 100" << endl;
            exit(1);}

    amount = d * 100 + c;
    if (s == minus) amount = -amount;
}

bool Currency::Set(sign s, unsigned long d,
                  unsigned int c)
{// 取值
    if (c > 99) return false;

    amount = d * 100 + c;
    if (s == minus) amount = -amount;
    return true;
}

bool Currency::Set(float a)
{// 取值

```

```
sign sgn;
if (a < 0) {sgn = minus; a = -a;}
else sgn = plus;
amount = (a + 0.001) * 100;
if (sgn == minus) amount = -amount;
return true;
}
```

程序1-23 函数Add和Output的新代码

```
Currency Currency::Add(const Currency& x) const
{// 把x 累加至 *this.
    Currency y;
    y.amount = amount + x.amount;
    return y;
}
```

```
void Currency::Output() const
{//输出currency 的值
    long a = amount;
    if (a < 0) {cout << '-'; a = -a;}
    long d = a / 100; // 美元
    cout << '$' << d << ' ';
    int c = a - d * 100; // 美分
    if (c < 10) cout << "0";
    cout << c;
}
```

### 1.4.3 操作符重载

Currency类包含了几个与 C++ 标准操作符相类似的成员函数，例如，Add进行+操作，Increment进行+=操作。直接使用这些标准的 C++ 操作符比另外定义新的函数（如 Add，Increment）要自然得多。可以借助于操作符重载（operator overloading）的过程来使用+和+=。操作符重载允许扩充现有C++操作符的功能，以便把它们直接应用到新的数据类型或类。

程序1-24给出了把Add和Increment分别替换为+和+=的类描述。Output函数采用一个输出流的名字作为参数。这些变化仅需修改Add和Output的代码（见程序1-23）。程序1-25给出了修改后的代码。在这个程序还给出了重载C++流插入操作符<<的代码。

注意在Currency类中重载了流插入操作符，但没有定义相应的成员函数，而重载+和+=时则把它们定义为类成员。同样，也可以重载流抽取操作符>>而不需要把它定义为类成员。请注意，是函数Output支持了<<的重载。由于Currency对象的private成员对于非类成员函数来说不可访问（被重载的<<不是类成员，而+是），所以，重载<<的代码不能引用对象x的私有成员（在<<操作中x将被插入到输出流中）。特别地，下面的代码是错误的，因为成员amount是不可访问的。

```
// 重载<<
ostream& operator<< ( ostream& out, const Currency& x)
{ out << x.amount; return out; }
```

## 程序1-24 使用操作符重载的类定义

---

```

class Currency {
public:
    // 构造函数
    Currency(sign s = plus, unsigned long d = 0, unsigned int c = 0);
    // 析构函数
    ~Currency() {}
    bool Set(sign s, unsigned long d, unsigned int c);
    bool Set(float a);
    sign Sign() const
    {if (amount < 0) return minus;
     else return plus;}
    unsigned long Dollars() const
    {if (amount < 0) return (-amount) / 100;
     else return amount / 100;}
    unsigned int Cents() const
    {if (amount < 0)
        return -amount - Dollars() * 100;
     else return amount - Dollars() * 100;}
    Currency operator+(const Currency& x) const;
    Currency& operator+=(const Currency& x)
    {amount += x.amount; return *this;}
    void Output(ostream& out) const;
private:
    long amount;
};

```

---

## 程序1-25 + , Output和&lt;&lt;的代码

---

```

Currency Currency::operator+(const Currency& x) const
{// 把 x 累加至*this.
    Currency y;
    y.amount = amount + x.amount;
    return y;
}

void Currency::Output(ostream& out) const
{// 将currency 的值插入到输出流
    long a = amount;
    if (a < 0) {out << '-'; a = -a;}
    long d = a / 100; // 美元
    out << '$' << d << '.';
    int c = a - d * 100; // 美分
    if (c < 10) out << "0";
    out << c;
}

// 重载<<
ostream& operator<<(ostream& out, const Currency& x)

```

```
{x.Output(out); return out;}
```

程序1-26是程序1-20的另一个版本，它假定操作符都已经被重载，程序1-24和1-25的代码位于文件curr3.h之中。

程序1-26 操作符重载的应用

```
#include <iostream.h>
#include "curr3.h"

void main(void)
{
    Currency g, h(plus, 3, 50), i, j;
    g.Set(minus, 2, 25);
    i.Set(-6.45);
    j = h + g;
    cout << j << endl;
    i += h;
    cout << i << endl;
    j = i + g + h;
    cout << j << endl;
    j = (i+=g) + h;
    cout << j << endl;
    cout << i << endl;
}
```

#### 1.4.4 引发异常

诸如构造函数和Set函数这样的类成员在执行预定的任务时有可能会失败。在构造函数中处理错误条件的方法是退出程序，而在Set中则返回一个失败信号（false）给调用者。实际上可以通过引发异常来处理这些错误，以便在程序最合适的地方捕获异常并进行处理。为了引发异常，必须首先定义一个异常类，比如BadInitializers(见程序1-27)。

程序1-27 异常类BadInitializers

```
// 初始化失败
class BadInitializers {
public:
    BadInitializers() {}
};
```

我们可以修改程序1-21中Set函数的描述，使其返回void类型。也可以修改构造函数的代码以及程序1-28中定义的第一个Set函数的代码。其他的代码不做修改。

程序1-28 引发异常

```
Currency::Currency(sign s, unsigned long d, unsigned int c)
{
    // 创建一个Currency对象
    if (c > 99) throw BadInitializers();

    amount = d * 100 + c;
}
```

```

    if (s == minus) amount = -amount;
}

void Currency::Set(sign s, unsigned long d, unsigned int c)
{// 取值
    if (c > 99) throw BadInitializers();

    amount = d * 100 + c;
    if (s == minus) amount = -amount;
}

```

#### 1.4.5 友元和保护类成员

正如前面所指出的那样，一个类的 `private` 成员仅对于类的成员函数是可见的。在有些应用中，必须把对这些 `private` 成员的访问权授予其他的类和函数，做法是把这些类和函数定义为友元 (`friend`)。

在 `Currency` 类例子中 (见程序 1-24)，定义了一个成员函数 `Output` 以便于对操作符 `<<` 的重载。定义这个函数是必要的，因为如下函数：

```
ostream& operator <<(ostream& out, const Currency& x)
```

不能访问 `private` 成员 `amount`。我们可以把 `ostream& operator<<` 描述为 `Currency` 类的友元，从而避免定义附加的函数，这样就把 `Currency` 所有成员 (包括 `private` 成员及 `public` 成员) 的访问权都授予了该函数。为了产生友元，需要在 `Currency` 类描述中引入 `friend` 语句。为一致起见，总是把 `friend` 语句放在类标题语句中，如：

```

class Currency {
    friend ostream& operator<< (ostream&, const Currency&);
public:

```

有了这个友元，就可以使用程序 1-29 中的代码来重载操作符 `<<`。当 `Currency` 的 `private` 成员发生变化时，必须检查它的友元以便做出相应的变化。

程序 1-29 重载友元 `<<`

```

// 重载 <<
ostream& operator<<(ostream& out, const Currency& x)
{// 把currency 的值插入到输出流
    long a = x.amount;
    if (a < 0) {out << '-'; a = -a;}
    long d = a / 100; // 美元
    out << '$' << d << '.';
    int c = a - d * 100; // 美分
    if (c < 10) out << "0";
    out << c;
    return out;
}

```

稍后我们将看到如何从一个类 `B` 派生出另外一个类 `A`，此时类 `A` 被称为派生类 (`drived class`)，类 `B` 被称为基类 (`base class`)。派生类需要访问基类的部分或所有数据成员。为了便于传递这些访问权，C++ 提供了第三类成员——保护类成员 (`protected`)。保护类成员类似于私有成员，

区别在于派生类可以访问保护类成员。

用户应用程序可以访问的类成员应被声明为 public 成员，数据成员尽量不要定义为这种类型，其他成员应分成 private 和 protected 两部分。软件工程实践告诉我们，数据成员应尽量保持为 private 成员。通过增加保护类成员来访问和修改数据成员的值，派生类可以间接访问基类的数据成员。同时，可以修改基类的实现细节而不会影响派生类。

#### 1.4.6 增加 #ifndef, #define 和 #endif 语句

文件 curr1.h(或 curr3.h)的全部内容包含了 Currency 类的描述及实现细节。在文件头，必须放上如下语句：

```
#ifndef Currency_  
#define Currency_
```

而在文件尾需要放上语句：

```
#endif
```

这些语句确保 Currency 的代码仅被程序包含（include）和编译一次。建议你为本书中所提供的其他类定义也加上相应的语句。

### 练习

9. 1) 采用程序 1-15 中的描述，所能表示的最大和最小货币值分别是多少？假定用四个字节表示一个 long 型数据，用两个字节表示一个 int 型数据，则一个 unsigned long 数介于  $0 \sim 2^{32}-1$  之间，一个 unsigned int 数介于  $0 \sim 65535$  之间。

2) 采用程序 1-15 中的描述，把 dollars 和 cents 变成 int 型，此时所能表示的最大和最小货币值分别是多少？

3) 如果用函数 Add（见程序 1-18）来累加两个货币值，为了确保从 Currency 类型转换成 long int 类型时不会发生错误，a1 和 a2 最大可能的值应是多少？

10. 试扩充程序 1-15 中的 Currency 类，为该类添加如下的 public 成员函数：

1) Input()——从标准输入流中接收一个货币值，并把它返回给调用者。

2) Subtract(x)——从当前对象中减去对象 x 的值，并把结果返回。

3) Percent(x)——返回一个 Currency 对象，其值为当前对象的 x%，其中 x 是一个浮点数。

4) Multiply(x)——返回一个 Currency 对象，其值为当前对象乘以浮点数 x。

5) Devide(x)——返回一个 Currency 对象，其值为当前对象除以浮点数 x。

11. 采用程序 1-21 中的描述来完成练习 10。

12. 1) 采用程序 1-24 中的描述来完成练习 10。重载操作符 >>, -, %, \* 和 /。在重载操作符 >> 时，可把它定义成一个友元函数，不必专门定义一个 public 输入函数。

2) 利用重载赋值操作符 = 来替换两个 Set 函数。把一个整数赋值给一个 Currency 对象可用 operator=(int x) 来表示，它可用来替换第一个 Set 函数，其中 x 表示一个包含符号、美元和美分的整数。同样，operator=(float x) 可用来替换第二个 Set 函数。

## 1.5 测试与调试

### 1.5.1 什么是测试

如 1.1 节所示，正确性是一个程序最重要的属性。由于采用严格的数学证明方法来证明一

个程序的正确性是非常困难的（哪怕是一个很小的程序），所以我们想转而求助于程序测试（program test）过程来实施这项工作。所谓程序测试是指在目标计算机上利用输入数据，也称之为测试数据（test data）来实际运行该程序，把程序的实际行为与所期望的行为进行比较。如果两种行为不同，就可判定程序中存在问题。然而，不幸的是，即使两种行为相同，也不能够断定程序就是正确的，因为对于其他的测试数据，两种行为又可能不一样。如果使用了许多组测试数据都能够看到这两种行为是一样的，我们可以增加对程序正确性的信心。通过使用所用可能的测试数据，可以验证一个程序是否正确。然而，对于大多数实际的程序，可能的测试数据的数量太大了，不可能进行穷尽测试，实际用来测试的输入数据空间的子集称之为测试集（test set）。

例1-4 [二次方程求解] 一个关于变量  $x$  的二次函数形式如下：

$$ax^2 + bx + c$$

其中  $a, b, c$  的值是已知的，且  $a \neq 0$ 。 $3x^2 - 2x + 4$ 、 $-9x^2 - 7x$ 、 $3.5x^2 + 4$  以及  $5.8x^2 + 3.2x + 5$  都是二次函数的实例。 $5x + 3$  不是二次函数。

二次函数的根是指使函数的值为 0 的那些  $x$ 。例如，函数  $f(x) = x^2 - 5x + 6$  的根为 2 和 3，因为  $f(2) = f(3) = 0$ 。每个二次函数都会有两个根，这两个根可用如下公式给出：

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

对于函数  $f(x) = x^2 - 5x + 6$ ， $a=1$ ， $b=-5$ ， $c=6$ ，把  $a, b, c$  代入以上公式，可得：

$$\frac{5 \pm \sqrt{25 - 4 \cdot 1 \cdot 6}}{2} = \frac{5 \pm 1}{2}$$

所以  $f(x)$  的根是  $x=3$  和  $x=2$ 。

当  $d = b^2 - 4ac = 0$  时，所得到的两个根是一样的；当  $d > 0$  时，两个根不同且是实数；当  $d < 0$  时，两个根也不相同且为复数，此时，每个根都有一个实部（real）和一个虚部（imaginary），实部为  $-b/2a$ ，虚部为  $\sqrt{-d}$ 。复数根为“实部+虚部\* $i$ ”和“实部-虚部\* $i$ ”，其中  $i = \sqrt{-1}$ 。

函数 OutputRoots（见程序 1-30）计算并输出一个二次方程的根。我们不去试图对该函数的正确性进行形式化证明，而是希望通过测试来验证其正确性。对于该程序来说，所有可能的输入数据的数目实际上就是所有不同的三元组（ $a, b, c$ ）的数目，其中  $a \neq 0$ 。即使  $a, b$  和  $c$  都被限制为整数，所有可能的三元组的数目也是非常巨大，要想测试所有的三元组是不可能的。若整数的长度为 16 位， $b$  和  $c$  都有  $2^{16}$  种不同取值， $a$  有  $2^{16} - 1$  种不同取值（因为  $a$  不能为 0），所有不同三元组的数目将达到  $2^{32}$ （ $2^{16} - 1$ ）。如果目标计算机能按每秒钟 1 000 000 个三元组的速率进行测试，那么至少需要 9 年才能完成！如果使用一个更快的计算机，按每秒测试 1 000 000 000 个三元组的速度，也至少需要三天才能完成。所以一个实际使用的测试集仅是整个测试数据空间中的一个子集。

程序 1-30 计算并输出一个二次方程的根

```
template<class T>
void OutputRoots(T a, T b, T c)
// 计算并输出一个二次方程的根

    T d = b*b - 4*a*c;
```



```

if (d > 0) { // 两个实数根
    float sqrt_d = sqrt(d);
    cout << "There are two real roots "
        << (-b+sqrt_d)/(2*a) << " and "
        << (-b-sqrt_d)/(2*a)
        << endl;}
else if (d == 0)
    // 两个根相同
    cout << "There is only one distinct root "
        << -b/(2*a)
        << endl;
else // 复数根
    cout << "The roots are complex"
        << endl
        << "The real part is "
        << -b/(2*a) << endl
        << "The imaginary part is "
        << sqrt(-d)/(2*a) << endl;
}

```

如果使用数据  $(a, b, c) = (1, -5, 6)$  来进行测试，程序将输出2和3，程序的行为与期望的行为是一致的，因此可以推断对于该输入数据，程序是正确的。然而，使用一个适当的测试数据子集来验证所观察行为与所期望行为的一致性并不能证明对于所有的输入数据，程序都能够正确工作。

由于可以提供给一个程序的不同输入数据的数目一般都非常巨大，所以测试通常都被限制在一个很小的子集中进行。使用子集所完成的测试不能完全保证程序的正确性。所以，测试的目的不是去建立正确性认证，而是要暴露程序中的错误！必须选择能暴露程序中所存在错误的测试数据，不同的测试数据可以暴露程序中不同的错误。

例1-5 测试数据  $(a, b, c) = (1, -5, 6)$  可以使函数 `OutputRoots` 执行产生两个实数根的代码，如果输出了2和3，可以有一些信心地认为在本次测试中所执行的代码是正确的。注意，一段错误的代码也可能给出正确的结果。例如，如果在关于  $d$  的表达式中忽略  $a$ ，将其错误地写成：

```
T d = b * b - 4 * c;
```

$d$  的值与所测试的结果相同，因为  $a=1$ 。由于使用测试数据  $(1, -5, 6)$  未能执行完代码中的所有语句，故我们对尚未执行的语句还没有多大的信心。

测试集  $\{(1, -5, 6), (1, 3, 2), (2, 5, 2)\}$  仅可用来暴露 `OutputRoots` 前7行语句中存在的错误，因为这个测试集中的每个三元组仅需要执行代码的前7行语句。然而，测试集  $\{(1, -5, 6), (1, -8, 16), (1, 2, 5)\}$  可使 `OutputRoots` 中的每行语句都得到执行，所以该测试集将可以暴露较多的错误。

### 1.5.2 设计测试数据

在设计测试数据的时候，应当牢记：测试的目标是去披露错误。如果用来寻找错误的测试数据找不到错误，我们就可以有信心相信程序的正确性。为了弄清楚对于一个给定的测试数据，程序是否存在错误，首先必须知道对于该测试数据，程序的正确结果应是什么。

例1-6 对于二次方程求解的例子，可以用如下两种方法之一来给定任意测试数据时程序的正

确输出。第一种方法是，计算出所测试二次方程的根。例如，系数  $(a, b, c) = (1, -5, 6)$  的二次方程的根为2和3。对于测试数据  $(1, -5, 6)$ ，可以把程序所输出的根与2和3进行比较，以验证程序1-30的正确性。第二种可行的方法是把程序所产生的根代入二次函数以验证函数的值是否真为0。所以，如果程序输出的是2和3，可以计算出  $f(2) = 2^2 - 5 \cdot 2 + 6 = 0$ ,  $f(3) = 3^2 - 5 \cdot 3 + 6 = 0$ 。可以把这种验证方法用计算机程序来实现。对于第一种方法，测试程序输入三元组  $(a, b, c)$  和期望的根，然后把程序计算出的根与期望的根进行比较。对于第二种方法，可以编写代码来计算对于程序输出的根，二次函数的相应函数值，然后验证这个值是否为0。

可以采用下面的条件来计算任何候选的测试数据：

- 这个数据能够发现错误的潜力如何？
- 能否验证采用这个数据时程序的正确性？

设计测试数据的技术分为两类：黑盒法 (black box method) 和白盒法 (white box method)。在黑盒法中，考虑的是程序的功能，而不是实际的代码。在白盒法中，通过检查程序代码来设计测试数据，以便使测试数据的执行结果能很好地覆盖程序的语句以及执行路径。

### 1. 黑盒法

最流行的黑盒法是I/O 分类及因果图，本节仅探讨I/O分类。在这种方法中，输入数据和 / 或输出数据空间被分成若干类，不同类中的数据会使程序所表现出的行为有质的不同，而相同类中的数据则使程序表现出本质上类似的行为。二次方程求解的例子中有三种本质上不同的行为：产生复数根，产生实数根且不同，产生实数根且相同。可以根据这三种行为把输入空间分为三类。第一类中的数据将产生第一种行为；第二类中的数据将产生第二种行为；而第三类中的数据将产生第三种行为。一个测试集应至少从每一类中抽取一个输入数据。

### 2. 白盒法

白盒法基于对代码的考察来设计测试数据。对一个测试集最起码的要求就是使程序中的每一条语句都至少执行一次。这种要求被称为语句覆盖 (statement coverage)。对于二次方程求解的例子，测试集  $\{(1, -5, 6), (1, -8, 16), (1, 2, 5)\}$  将使程序中的每一条语句都得以执行，而测试集  $\{(1, -5, 6), (1, 3, 2), (2, 5, 2)\}$  则不能提供语句覆盖。

在分支覆盖 (decision coverage) 中要求测试集要能够使程序中的每一个条件都分别能出现true和false两种情况。程序1-30中的代码有两个条件： $d > 0$ 和 $d == 0$ 。在进行分支覆盖测试时，要求测试集至少能使条件 $d > 0$ 和 $d == 0$ 分别出现一次为true、一次为false的情况。

例1-7 [求最大元素] 程序1-31用于返回数组 $a[0:n-1]$ 中最大元素所在的位置。它依次扫描 $a[0]$ 到 $a[n-1]$ ，并用变量pos来保存到目前为止所能找到的最大元素的位置。数据集  $a[0:4] = [2, 4, 6, 8, 9]$  能够提供语句覆盖，但不能提供分支覆盖，因为条件 $a[pos] < a[i]$ 不会变成false。数据集 $[4, 2, 6, 8, 9]$ 既能提供语句覆盖也能提供分支覆盖。

程序1-31 寻找最大元素

```
template<class T>
int Max(T a[], int n)
// 寻找 a[0:n-1]中的最大元素
{
    int pos = 0;
    for (int i = 1; i < n; i++)
        if (a[pos] < a[i])
            pos = i;
}
```

```
return pos;
}
```

可以进一步加强分支覆盖的条件,要求每个条件中的每个从句 ( clause ) 既能出现true也能出现false的情况,这种加强的条件被称之为从句覆盖 ( clause coverage )。一个从句在形式上被定义成一个不包含布尔操作符 ( 如 &&,||,! ) 的布尔表达式。表达式  $x > y$ ,  $x + y < y * z$  以及  $c$  ( $c$  是一个布尔类型) 都是从句的例子。考察如下语句:

```
if((C1 && C2) || (C3 && C4)) S1;
else S2;
```

其中  $C1$ ,  $C2$ ,  $C3$  和  $C4$  是从句,  $S1$  和  $S2$  是语句。在分支覆盖方式下,需要使用一个能使  $((C1 \&\& C2) || (C3 \&\& C4))$  为true的测试数据以及一个能使该条件为 false 的测试数据。而从句覆盖则要求测试数据能使四个从句  $C1, C2, C3$  和  $C4$  都分别至少取一次 true 值和至少取一次 false 值。

还可以继续加强从句覆盖的条件,要求测试各从句值的所有可能组合。对于上面的条件  $((C1 \&\& C2) || (C3 \&\& C4))$ , 加强后的从句覆盖要求使用 16 个测试数据集: 每个测试集对应于四个从句值组合后的情形。不过, 其中有些组合是不可能的。

如果按照某个测试数据集来排列程序语句的执行次序,可以得到一条执行路径 ( execution path )。不同的测试数据可能会得到不同的执行路径。程序 1-30 仅存在三条执行路径——第 1 行至第 7 行, 第 1、2、8~12 行, 第 1、2、8、13~19 行。而程序 1-31 中的执行路径则随着  $n$  的增加而增加。当  $n=1$  时, 仅有一条执行路径——1、2、5 行; 当  $n=2$  时, 有两条路径——1、2、3、2、5 和 1、2、3、4、2、5 行; 当  $n=3$  时, 有四条路径——1、2、3、2、3、2、5 行, 1、2、3、4、2、3、2、5 行, 1、2、3、2、3、4、2、5 行, 1、2、3、4、2、3、4、5 行。执行路径覆盖要求测试数据集能使每条执行路径都得以执行。对于二次方程求解程序, 语句覆盖、分支覆盖、从句覆盖以及执行路径覆盖都是等价的, 但对于程序 1-31, 语句覆盖、分支覆盖、和执行路径覆盖是不同的, 而分支覆盖和从句覆盖是等价的。

在这些白盒测试方法中, 一般要求实现执行路径覆盖。一个能实现全部执行路径覆盖的测试数据同样能实现语句覆盖和分支覆盖, 然而, 它可能无法实现从句覆盖。全部执行路径覆盖通常会需要无数的测试数据或至少是非常可观的测试数据, 所以在实践中一般不可能进行全部执行路径覆盖。

本书中的许多练习都要求你测试所编代码的正确性。你所使用的测试数据应至少提供语句覆盖。此外, 你必须测试那些可能会使你的程序出错的特定情形。例如, 对于一个用来对  $n$  个元素进行排序的程序, 除了测试  $n$  的正常取值外, 还必须测试  $n=0, 1$  这两种特殊情形。如果该程序使用数组  $a[0:99]$ , 还需要测试  $n=100$  的情形。  $n=0, 1$  和 100 分别表示边界条件为空, 单值和全数组的情形。

### 1.5.3 调试

测试能够发现程序中的错误。一旦测试过程中产生的结果与所期望的结果不同, 就可以了解到程序中存在错误。确定并纠正程序错误的过程被称为调试 ( debug )。尽管透彻地研究程序调试的方法超出了本书的范围, 但我们还是提供一些好的建议给大家:

- 可以用逻辑推理的方法来确定错误语句。如果这种方法失败, 还可以进行程序跟踪, 以确定程序什么时候开始出现错误。如果对于给定的测试数据程序需要运行很多指令, 因而需要跟踪太多语句, 很难人工确定错误, 此时, 这种方法就不太可行了, 在这种情况下, 必须试着把可疑的代码分离出来, 专门跟踪这段代码。

- 不要试图通过产生异常来纠正错误。异常的数量可能会迅速增长。必须首先找到需要纠正的错误，然后根据需要重新设计。

- 在纠正一个错误时，必须保证不会产生一个新的、以前没有的错误。用原本能使程序正确运行的测试数据来运行纠正过错误的程序，确信对于该数据，程序仍然正确。

- 在测试和调试一个有错的程序时，从一个与其他函数独立的函数开始。这个函数应该是一个典型的输入或输出函数。然后每次引入一个尚未测试的函数，测试并调试更大一些的程序。这种策略被称为增量测试与调试（incremental test and debug）。在使用这种策略时，可以有理由认为产生错误的语句位于刚刚引入的函数之中。

## 练习

13. 证明能够为程序 1-30 提供语句覆盖的测试集也能提供分支覆盖和执行路径覆盖。

14. 为程序 1-31 设计一个  $n=4$  的测试数据集，要求该测试集能提供执行路径覆盖。

15. 程序 1-8 中有多少条执行路径？

16. 程序 1-9 中有多少条执行路径？

## 1.6 参考及推荐读物

1) J.Cohoon, J.Davidson. *C++ Program Design: An Introduction to Programming and Object-Oriented Design*. Richard D.Irwin, 1997。

2) H.Deitel, P.Deitel. *C++ How to Program*. Prentice Hall, 1994。

以上两本书是比较好的 C++ 语言入门教材。

3) G.Myers. *The Art of Software Testing*. John Wiley, 1979。

4) Boris Beizer. *Software Testing Techniques* 第2版. Van Nostrand Reinhold, 1990。

后两本书对于软件测试及调试技术有更透彻的介绍。

China-pub.com

下载

## 第2章 程序性能

以下是本章中所介绍的有关程序性能分析与测量的概念：

- 确定一个程序对内存及时间的需求。
- 使用操作数和执行步数来测量一个程序的时间需求。
- 采用渐进符号描述复杂性，如  $O$ 、 $\Omega$ 、 $\Theta$ 、 $o$ 。
- 利用计时函数测量一个程序的实际运行时间。

除了上述概念以外，本章还给出了许多应用代码，在后续章节中将可以看到，这些代码有很多用处。这些应用包括：

- 在一个数组中搜索具有指定特征的元素。本章中所使用的方法是顺序搜索和折半搜索。
- 对数组元素进行排序。本章给出了计数排序、选择排序、冒泡排序及插入排序的实现代码。
- 采用Horner 法则计算一个多项式。
- 执行矩阵运算，如矩阵加、矩阵转置和矩阵乘。

### 2.1 引言

所谓程序性能（program performance），是指运行一个程序所需要的内存大小和时间。可以采用两种方法来确定一个程序的性能，一个是分析的方法，一个是实验的方法。在进行性能分析（performance analysis）时，采用分析的方法，而在进行性能测量（performance measurement）时，借助于实验的方法。

程序的空间复杂性（space complexity）是指运行完一个程序所需要的内存大小。对一个程序的空间复杂性感兴趣的主要原因如下：

- 如果程序将要运行在一个多用户计算机系统中，可能需要指明分配给该程序的内存大小。
- 对任何一个计算机系统，想提前知道是否有足够可用的内存来运行该程序。
- 一个问题可能有若干个内存需求各不相同的解决方案。比如，对于你的计算机来说，某个C++编译器仅需要1MB的空间，而另一个C++编译器可能需要4MB的空间。如果你的计算机中内存少于4MB，你只能选择1MB的编译器。如果较小编译器的性能比得上较大的编译器，即使用户的计算机中有额外的内存，也宁愿使用较小的编译器。

- 可以利用空间复杂性来估算一个程序所能解决问题的最大规模。例如，有一个电路模拟程序，用它模拟一个有  $c$  个元件、 $w$  个连线的电路需要  $280K + 10 * (c + w)$  字节的内存。如果可利用的内存总量为640K字节，那么最大可以模拟  $c + w \leq 36K$  的电路。

程序的时间复杂性（time complexity）是指运行完该程序所需要的时间。对一个程序的时间复杂性感兴趣的主要原因如下：

- 有些计算机需要用户提供程序运行时间的上限，一旦达到这个上限，程序将被强制结束。一种简易的办法是简单地指定时间上限为几千年。然而这种办法可能会造成严重的财政问题，因为如果由于数据问题导致你的程序进入一个死循环，你可能需要为你所使用的机时付出巨额资金。因此我们希望能提供一个稍大于所期望运行时间的的时间上限。

- 正在开发的程序可能需要提供一个满意的实时响应。例如，所有交互式程序都必须提供实时响应。一个需要1分钟才能把光标上移一页或下移一页的文本编辑器不可能被众多的用户接受；一个电子表格程序需要花费几分钟才能对一个表单中的单元进行重新计值，那么只有非常耐心的用户才会乐意使用它；如果一个数据库管理系统在对一个关系进行排序时，用户可以有时间去喝两杯咖啡，那么它也很难被用户接受。为交互式应用所设计的程序必须提供满意的实时响应。根据程序或程序模块的时间复杂性，可以决定其响应时间是否可以接受，如果不能接受，要么重新设计正在使用的算法，要么为用户提供一台更快的计算机。

- 如果有多种可选的方案来解决一个问题，那么具体决定采用哪一个主要基于这些方案之间的性能差异。对于各种解决方案的时间及空间复杂性将采用加权的方式进行评价。

## 练习

1. 给出两种以上的原因说明为什么程序分析员对程序的空间复杂性感兴趣？
2. 给出两种以上的原因说明为什么程序分析员对程序的时间复杂性感兴趣？

## 2.2 空间复杂性

### 2.2.1 空间复杂性的组成

程序所需要的空间主要由以下部分构成：

- 指令空间（instruction space） 指令空间是指用来存储经过编译之后的程序指令所需的

空间。

- 数据空间（data space） 数据空间是指用来存储所有常量和所有变量值所需的

空间。数据空间由两个部分构成：

- 1) 存储常量（见程序1-1至1-9中的数0、1和4）和简单变量（见程序1-1至1-6中的a、b和c）所需要的空间。

- 2) 存储复合变量（见程序1-8和1-9中的数组a）所需要的空间。这一类空间包括数据结构所需的

空间及动态分配的空间。

- 环境栈空间（environment stack space） 环境栈用来保存函数调用返回时恢复运行所需要的信息。例如，如果函数fun1调用了函数fun2，那么至少必须保存fun2结束时fun1将要继续执行的指令的地址。

#### 1. 指令空间

程序所需要的指令空间的数量取决于如下因素：

- 把程序编译成机器代码的编译器。
- 编译时实际采用的编译器选项。
- 目标计算机。

在决定最终代码需要多少空间的时候，编译器是一个最重要的因素。图2-1给出了用来计算表达式 $a+b+b*c+(a+b-c)/(a+b)+4$ 的三段可能的代码，它们都执行完全相同的算术操作（如，每个操作符都有相同的操作数），但每段代码所需要的空间都不一样。所使用的编译器将确定产生哪一种代码。



LOAD	a	LOAD	a	LOAD	a
ADD	b	ADD	b	ADD	b
STORE	t1	STORE	t1	STORE	t1
LOAD	b	SUB	c	SUB	c
MULT	c	DIV	t1	DIV	t1
STORE	t2	STORE	t2	STORE	t2
LOAD	t1	LOAD	b	LOAD	b
ADD	t2	MUL	c	MUL	c
STORE	t3	STORE	t3	ADD	t2
LOAD	a	LOAD	t1	ADD	t1
ADD	b	ADD	t3	ADD	4
SUB	c	ADD	t2		
STORE	t4	ADD	4		
LOAD	a				
ADD	b				
STORE	t5				
LOAD	t4				
DIV	t5				
STORE	t6				
LOAD	t3				
ADD	t6				
ADD	4				
a)		b)		c)	

图2-1 三段等价的代码

即使采用相同的编译器，所产生程序代码的大小也可能不一样。例如，一个编译器可能为用户提供优化选项，如代码优化以及执行时间优化等。比如，在图 2-1 中，在非优化模式下，编译器可以产生图 2-1b 的代码。在优化模式下，编译器可以利用知识  $a+b+b*c=b*c+(a+b)$  来产生图 2-1c 中更短、更高效的代码。使用优化模式通常会增加程序编译所需要的时间。

从图 2-1 的例子中可以看到，一个程序还可能需要其他额外的空间，即诸如临时变量  $t1, t2, \dots, t6$  所占用的空间。

另外一种可以显著减少程序空间的编译器选项就是覆盖选项，在覆盖模式下，空间仅分配给当前正在执行的程序模块。在调用一个新的模块时，需要从磁盘或其他设备中读取，新模块的代码将覆盖原模块的代码。所以程序空间就等价于最大的模块所需要的空间（而不是所有模块之和）。

目标计算机的配置也会影响代码的规模。如果计算机具有浮点处理硬件，那么每个浮点操作可以转换成一条机器指令。如果没有安装浮点处理硬件，必须生成仿真的浮点计算代码。

## 2. 数据空间

对于简单变量和常量来说，所需要的空间取决于所使用的计算机和编译器以及变量与常量的数目。之所以如此是因为我们通常关心所需内存的字节数。由于每字节所占用的位数依赖于具体的机器环境，因此每个变量所需要的空间也会有所不同。此外，存储  $2^{100}$  也将比存储  $2^3$  需要更多的位数。

图2-2中列出了Borland C++中每种简单变量所占用的空间。对于一个结构变量，可以把它的每个成员所占用的空间累加起来即可得到该变量所需要的内存。类似地，可以得到一个数组变量所需要的空间，方法是用数组的大小乘以单个数组元素所需要的空间。

类 型	占用字节数	范 围
char	1	-128~127
unsigned char	1	0~255
short	2	-32 768~32 767
int	2	-32 768~32 767
unsigned int	2	0~65 535
long	4	$-2^{31} \sim 2^{31}-1$
unsigned long	4	$0 \sim 2^{32}-1$
float	4	$\pm 3.4\text{E} \pm 38$
double	8	$\pm 1.7\text{E} \pm 308$
long double	10	$3.4\text{E}-4932 \sim 1.1\text{E}+4932$
pointer	2	(near, _cs, _ds, _es, _ss 指针)
pointer	4	(far, huge 指针)

注意：在32位Borland C++程序中，int类型的长度为4

图2-2 Borland C++中每种简单变量所占用的空间（摘自 Borland C++ Programmer's Guide，Borland 公司，加州Scotts Valley, 1996）

考察如下的数组定义：

```
double a[100];
int maze[rows][cols];
```

数组a 需要的空间为100个double类型元素所占用的空间，若每个元素占用8个字节，则分配给该数组的空间总量为800字节。数组maze有rows\*cols个int类型的元素，它所占用的总空间为2\*rows\*cols字节。

### 3. 环境栈

在刚开始从事性能分析时，分析员通常会忽略环境栈所需要的空间，因为他们不理解函数是如何被调用的以及在函数调用结束时会发生什么。每当一个函数被调用时，下面的数据将被保存在环境栈中：

- 返回地址。
- 函数被调用时所有局部变量的值以及传值形式参数的值（仅对于递归函数而言）。
- 所有引用参数及常量引用参数的定义。

每当递归函数Rsum(见程序1-9)被调用时，不管该调用是来自外部或第4行，a的当前赋值、n的值以及程序运行结束时的返回地址都被存储在环境栈之中。

值得注意的是，有些编译器在保留局部变量的值、传值形式参数的值以及引用参数和常量引用参数的定义时，对于递归函数和非递归函数一视同仁，而有些编译器则仅为递归函数保存上述内容。所以实际使用的编译器将影响环境栈所需要的空间。

### 4. 小结

程序所需要的空间取决于多种因素，有些因素在构思或编写程序时是未知的（如将要使用

的计算机及编译器)，不过即使这些因素已经完全确定，我们也无法精确地分析一个程序所需要的空间。

然而，我们可以确定程序中某些部分的空间需求，这些部分依赖于所解决实例的特征。一般来说，这些特征包含决定问题规模的那些因素（如，输入和输出的数量或相关数的大小）。例如，对于一个对  $n$  个元素进行排序的程序，可以确定该程序所需要的空间为  $n$  的函数；对于一个累加两个  $n \times n$  矩阵的程序，可以使用  $n$  作为其实例特征；而对于累加两个  $m \times n$  矩阵的程序，可以使用  $m$  和  $n$  作为实例特征。

指令空间的大小对于所解决的特定问题不够敏感。常量及简单变量所需要的数据空间也独立于所解决的问题，除非相关数的大小对于所选定的数据类型来说实在太太，这时，要么改变数据类型，要么使用多精度算法重写该程序，然后再对新程序进行分析。

复合变量及动态分配所需要的空间同样独立于问题的规模。而环境栈通常独立于实例的特征，除非正在使用递归函数。在使用递归函数时，实例特征通常（但不总是）会影响环境栈所需要的空间数量。

递归函数所需要的栈空间通常称之为递归栈空间（recursion stack space）。对于每个递归函数而言，该空间主要依赖于局部变量及形式参数所需要的空间。除此以外，该空间还依赖于递归的深度（即嵌套递归调用的最大层次）。在程序 1-9 中嵌套递归调用一直进行到  $n=0$ ，这时，嵌套调用的层次关系如图 2-3 所示。该程序的最大递归深度为  $n+1$ 。

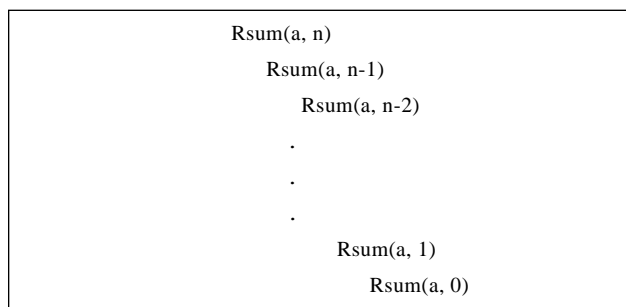


图2-3 程序1-9的嵌套调用层次

至此，可以把一个程序所需要的空间分成两部分：

- 固定部分，它独立于实例的特征。一般来说，这一部分包含指令空间（即代码空间）、简单变量及定长复合变量所占用空间、常量所占用空间等等。

- 可变部分，它由以下部分构成：复合变量所需的空间（这些变量的大小依赖于所解决的具体问题），动态分配的空间（这种空间一般都依赖于实例的特征），以及递归栈所需的空间（该空间也依赖于实例的特征）。

任意程序  $P$  所需要的空间  $S(P)$  可以表示为：

$$S(P) = c + S_p(\text{实例特征})$$

其中  $c$  是一个常量，表示固定部分所需要的空间， $S_p$  表示可变部分所需要的空间。一个精确的分析还应当包括在编译期间所产生的临时变量所需要的空间（如图 2-1 所示），这种空间是与编译器直接相关的，除依赖于递归函数外，它还依赖于实例的特征。本书将忽略这种空间。

在分析程序的空间复杂性时，我们将把注意力集中在估算  $S_p$ （实例特征）上。对于任意给

定的问题，首先需要确定实例的特征以便于估算空间需求。实例特征的选择是一个很具体的问题，我们将求助于介绍各种可能性的实际例子。一般来说，我们的选择受到相关数的数量以及程序输入和输出的规模的限制。有时还会使用更复杂的估算数据，这些数据来自于数据项之间的相互关系。

### 2.2.2 举例

例2-1 考察程序1-4。在估算 $S_p$ 之前，必须选择分析所使用的实例特征。两种可能性是：(1)数据类型T；(2) a, b 和c 的大小。假定使用T作为实例特征。由于a, b和c是引用参数，所以在函数中不需要为它们的值分配空间，但是必须保存指向这些参数的指针。如果每个指针需要 2个字节，那么共需要6个字节的指针空间。因此函数所需要的总空间是一个常量， $S_{Abc}$  (实例特征)=0。如果函数Abc 的参数是传值参数，那么每个参数需要分配大小为 sizeof (T) 的空间。在本例中，a, b 和c 所需要的空间为  $3 * \text{sizeof} (T)$ 。所需要的其他空间都独立于 T。因此 $S_{Abc}$  (实例特征)= $3 * \text{sizeof} (T)$ 。如果使用a, b 和c 的大小作为实例特征，则不管使用引用参数还是使用传值参数，都有 $S_{Abc}$  (实例特征)=0。注意在传值参数的情形下，分配给每个 a,b和c的空间均为 sizeof(T)，而不考虑存储在这些变量中的实际值是多大。例如，如果T是double类型，那么每个字节将被分配8个字节的空間。

例2-2 [顺序搜索] 程序2-1从左至右检查数组a[0:n-1]中的元素，以查找与x相等的那些元素。如果找到一个元素与x 相等，则函数返回x 第一次出现所在的位置。如果在数组中没有找到这样的元素，函数返回-1。

程序2-1 顺序搜索

```
template<class T>
int SequentialSearch(T a[], const T& x, int n)
{// 在未排序的数组 a[0:n-1]中搜索 x
// 如果找到，则返回所在位置，否则返回 - 1
int i;
for (i = 0; i < n && a[i] != x; i++);
if (i == n) return -1;
return i;
}
```

我们希望采用实例特征n 来估算该函数的空间复杂性。假定T 为int 类型，则数组a 中的每个元素需要2个字节，实参x需要2个字节，传值形式参数n 需要2个字节，局部变量i 需要2个字节，每个整型常量0和-1也分别需要2个字节。因此，所需要的总的的数据空间为12字节。因为该空间独立于n，所以 $S_{\text{顺序搜索}}(n) = 0$ 。

注意数组a 必须足够大以容纳所查找的n 个元素。不过，该数组所需要的空间已在定义实际参数（对应于a）的函数中分配，所以，不需要把该数组所需要的空间加到函数SequentialSearch所需要的空间上去。

例2-3 考察函数Sum（见程序1-8）。假定我们有兴趣把该函数所需要的空间看成欲累加元素的总数的函数。在该函数中，a, n, i 和tsum 需要分配空间，所以程序所需要的空间与n 无关，因此有 $S_{\text{Sum}}(n) = 0$ 。

例2-4 考察函数Rsum (见程序1-9)。如上例一样,假定实例特征为  $n$ 。递归栈空间包括形式参数  $a$  和  $n$  以及返回地址的空间。对于  $a$ , 需要保留一个指针, 而对于  $n$  则需要保留一个 `int` 类型的值。如果假定指针为 `near` 指针, 则该指针需要2个字节的存储空间。如果同时假定返回地址也占用2个字节, 那么根据 `int` 类型需要2个字节的常识, 可以确定每一次调用 Rsum 需要6个字节的栈空间。由于递归的深度为  $n+1$ , 所以需要  $6(n+1)$  字节的递归栈空间, 因而  $S_{Rsum}(n) = 6(n+1)$ 。

程序1-8所需要的空间比程序1-9所需要空间要小。

例2-5 [阶乘运算] 通过分析程序1-7中计算阶乘的函数可知, 该程序的空间复杂性是  $n$  的函数而不是输入 (只有一个) 或输出 (也只有一个) 个数的函数。递归深度为  $\max\{n, 1\}$ 。每次调用函数Factorial 时, 递归栈需要保留返回地址 (2个字节) 和  $n$  的值 (2个字节)。此外没有其他依赖于  $n$  的空间, 所以  $S_{Factorial}(n) = 4 * \max\{n, 1\}$ 。

例2-6 [排列方式] 程序1-10输出一组元素的所有排列方式。对于初始调用 Perm (list, 0,  $n-1$ ), 递归的深度为  $n$ 。由于每次调用需要10个字节的递归栈空间 (每个返回地址、list、 $k$ 、 $m$  以及  $i$  各需要2个字节), 所以需要  $10n$  字节的递归栈空间,  $S_{Perm}(n) = 10n$ 。

## 练习

3. 试采用两种C++编译器编译同一个C++程序, 所得代码的长度相同吗?

4. 给出可能影响程序空间复杂性的其他因素。

5. 使用图2-2所提供的数据来计算如下数组所需要的字节数:

1) `int matrix[10][100]`

2) `double x[100][5][20]`

3) `long double y[3]`

4) `float z[10][10][10][5]`

5) `short z[2][3][4]`

6) `long double b[3][3][3][3]`

6. 程序2-2给出了一个在数组  $a[0:n-1]$  中查找元素  $x$  的递归函数。如果找到  $x$ , 则函数返回  $x$  在  $a$  中的位置, 否则返回 -1。试计算  $S_p(n)$ 。

程序2-2 执行顺序搜索的递归函数

```
template <class T>
int SequentialSearch(T a[], const T& x, int n)
{//在未排序的数组a[0:n-1]中查找x
//如果找到则返回所在位置, 否则返回 - 1
if (n < 1) return -1;
if (a[n-1] == x) return n-1;
return SequentialSearch(a, x, n-1);
}
```

7. 编写一个非递归函数计算  $n!$  (例1-1), 并比较该函数的空间复杂性与程序1-7中递归函数的空间复杂性。

## 2.3 时间复杂性

### 2.3.1 时间复杂性的组成

影响一个程序空间复杂性的因素也都能影响程序的时间复杂性。一个程序在一台每秒钟能执行 $10^9$ 条指令的机器上运行要比在每秒仅能执行 $10^6$ 条指令的机器上快得多。图2-1c中的代码要比图2-1a中的代码运行时间更少。较小的问题所需要的运行时间通常要比较大的问题需要的时间少。

一个程序 $P$ 所占用的时间 $T(P)$ =编译时间+运行时间。编译时间与实例的特征无关。另外，可以假定一个编译过的程序可以运行若干次而不需要重新编译。因此我们将主要关注程序的运行时间。运行时间通常用“ $t_p$  (实例特征)”来表示。

由于在构思一个程序时，影响 $t_p$ 的许多因素还是未知的，所以我们有理由仅仅对 $t_p$ 进行估算。如果我们了解所用编译器的特征，就可以确定代码 $P$ 进行加、减、乘、除、比较、读、写等所需要的时间，从而可以得到一个计算 $t_p$ 的公式。令 $n$ 代表实例的特征，可以得到如下形式的表达式：

$$t_p(n) = c_a ADD(n) + c_s SUB(n) + c_m MUL(n) + c_d DIV(n) + \dots$$

其中 $c_a$ ,  $c_s$ ,  $c_m$ 和 $c_d$ 分别表示一个加、减、乘和除操作所需要的时间，函数 $ADD$ 、 $SUB$ 、 $MUL$ 和 $DIV$ 分别表示代码 $P$ 中所使用的加、减、乘和除操作的次数。

存在这样一个事实：一个算术操作所需要的时间取决于操作数的类型（int, float, double等），这个事实增加了获得一个精确的计算公式的烦琐程度。所以必须按照数据类型对操作进行分类。

有两个更可行的方法可用来估算运行时间：1) 找出一个或多个关键操作，确定这些关键操作所需要的执行时间；2) 确定程序总的执行步数。

### 2.3.2 操作计数

估算一个程序或函数的时间复杂性的一种方式就是首先选择一种或多种操作（如加、乘和比较等），然后确定这种(些)操作分别执行了多少次。这种方法是否成功取决于识别关键操作的能力，这些关键操作对时间复杂性的影响最大。下面给出的几个例子都采用了这种方法。

**例2-7 [最大元素]** 程序1-31返回数组 $a[0:n-1]$ 中最大元素的位置。我们可以根据数组元素之间所进行的比较数目来估算其时间复杂性。for循环中的每一次循环都需要执行一次这样的比较，所以总的比较次数为 $n-1$ 。函数Max还执行了其他的比较（for循环中的每一次循环之前都要比较一下 $i$ 和 $n$ ），这些比较没有包含在上述估算之中。其他的操作，比如初始化 $pos$ 以及循环控制变量 $i$ 的每次增值也没有包含在估算之中。如果把这些操作都纳入计数，则操作计数将增加一个常量。

**例2-8 [多项式求值]** 考察多项式 $P(x) = \sum_{i=0}^n c_i x^i$ 。如果 $c_n \neq 0$ ，则 $P$ 是一个 $n$ 维多项式。程序2-3可用来计算对于给定的值 $x$ ， $P(x)$ 的实际取值。假定根据for循环内部所执行的加和乘的次数来估算时间复杂性。可以使用维数 $n$ 作为实例特征。进入for循环的总次数为 $n$ ，每次循环执行1次加法和2次乘法（这种操作计数不包含循环控制变量 $i$ 每次递增所执行的加法）。加法的次数为 $n$ ，乘法的次数为 $2n$ 。



程序2-3 对多项式进行求值的函数

---

```
template <class T>
T PolyEval(T coeff[], int n, const T& x)
{//计算n次多项式的值，coeff[0:n]为多项式的系数
    T y=1, value= coeff[0];
    for ( int i = 1; i <= n; i++)
        { //累加下一项
            y *= x;
            value += y * coeff[i];
        }
    return value;
}
```

---

Horner 法则采用如下的分解式计算一个多项式：

$$P(x) = (\dots((c_n * x + c_{n-1}) * x + c_{n-2}) * x + c_{n-3}) * x + \dots) * x + c_0$$

相应的C++函数见程序2-4。采用与程序2-3相同的方法，可以估算出该程序的时间复杂性为 $n$ 次加法和 $n$ 次乘法。由于函数PolyEval所执行的加法数与Horner相同，而乘法数是Horner的两倍，因此，函数Horner应该更快。

程序2-4 利用Horner 法则对多项式进行求值

---

```
template <class T>
T Horner(T coeff[], int n, const T& x)
{//计算n次多项式的值，coeff[0:n]为多项式的系数
    T value= coeff[n];
    for( int i = 1; i <= n; i++)
        value = value * x + coeff[n-i];
    return value;
}
```

---

例2-9 [计算名次] 元素在队列中的名次 (rank) 可定义为队列中所有比它小的元素数目加上在它左边出现的与它相同的元素数目。例如，给定一个数组  $a=[4, 3, 9, 3, 7]$  作为队列，则各元素的名次为  $r=[2, 0, 4, 1, 3]$ 。函数Rank (见程序2-5) 可用来计算数组  $a[0:n-1]$  中各元素的名次。可以根据  $a$  的元素之间所进行的比较操作来估算程序的时间复杂性。这些比较操作是由 if 语句来完成的。对于  $i$  的每个取值，执行比较的次数为  $i$ ，所以总的比较次数为  $1+2+3+\dots+n-1 = (n-1)n/2$ 。

程序2-5 计算名次

---

```
template <class T>
void Rank(T a[], int n, int r[])
{//计算a[0:n-1]中n个元素的排名
    for ( int i = 1; i < n; i++)
        r[i] = 0; //初始化
    //逐对比较所有的元素
    for ( int i = 1; i < n; i++)
        for ( int j = 1; j < i; j++)
```

---



```
    if (a[j] <= a[i]) r[i]++;  
    else r[j]++;  
}
```

注意在估算时间复杂性时没有考虑 for 循环的额外开销、初始化数组  $r$  的开销以及每次比较  $a$  中两个元素时对  $r$  进行增值的开销。

例2-10 [按名次排序] 一旦使用程序2-5计算出数组中每个元素的名次，就可以利用元素名次按照递增的次序对数组中的元素进行重新排列，使得  $a[0] \ a[1] \ \dots \ a[n-1]$ 。如果能够使用一个附加的数组  $u$ ，那么可以采用程序2-6中给出的函数 `Rearrange` 来重新排列元素的次序。

在函数 `Rearrange` 执行期间移动元素的次数为  $2n$ 。（练习11要求怎样才能将移动次数减至  $n$ ）。完成整个排序需要执行  $(n-1)n/2$  次比较操作和  $2n$  次移动操作。这种排序方法被称为计数排序 (rank sort)。另外一种重排元素的函数见程序2-11，该函数没有使用附加数组  $u$ 。

程序2-6 利用附加数组重排数组元素

```
template <class T>  
void Rearrange (T a[], int n, int r[])  
{//按序重排数组 a 中的元素，使用附加数组 u  
    T *u = new T[n+1];  
    //在 u 中移动到正确的位置  
    for (int i = 1; i < n; i++)  
        u[r[i]] = a[i];  
    //移回到 a 中  
    for (int i = 1; i < n; i++)  
        a[i] = u[i];  
    delete [] u;  
}
```

例2-11 [选择排序] 例2-10给出了一种按递增次序重排数组  $a$  中元素的方法。另外一种可选的方法是：首先找出最大的元素，把它移动到  $a[n-1]$ ，然后在余下的  $n-1$  个元素中寻找最大的元素并把它移动到  $a[n-2]$ ，如此进行下去，这种排序方法为选择排序 (selection sort)，程序2-7中给出了实现这一过程的 C++ 函数 `SelectionSort`，其中函数 `Max` 在程序1-31中已经给出。可以按照元素的比较次数来估算函数的时间复杂性。从例2-7中已经知道每次调用 `Max(a, size)` 需要执行  $size-1$  次比较，所以总的比较次数为  $1+2+3+\dots+n-1=(n-1)n/2$ 。元素的移动次数为  $3(n-1)$ 。选择排序所需要的比较次数与按名次排序（见程序2-10）所需要的比较次数相同，但所需要的元素移动次数多出50%。本节的后面将介绍另外一种选择排序的方法。

程序2-7 选择排序

```
template <class T>  
void SelectionSort (T a[], int n)  
{//对数组 a[0:n-1] 中的 n 个元素进行排序  
    for (int size = n; size > 1; size--) {  
        int j = Max(a, size);  
        Swap(a[j], a[size-1]);  
    }  
}
```

例2-12 [冒泡排序] 冒泡排序 (bubble sort) 是另一种简单的排序方法。这种排序采用一种“冒泡策略”把最大元素移到右部。在冒泡过程中, 对相邻的元素进行比较, 如果左边的元素大于右边的元素, 则交换这两个元素。假定我们有四个元素 [5,3,7,1]。首先对5和3进行比较并交换, 得到 [3,5,7,1], 然后对5和7进行比较, 两者无须交换, 接下来比较7和1并交换, 得到 [3,5,1,7]。在一次冒泡过程结束后, 可以确信最大的元素肯定在最右边的位置上。函数 Bubble(见程序2-8)执行一次冒泡过程, 其中元素比较的次数为  $n-1$ 。

程序2-8 一次冒泡

---

```
template <class T>
void Bubble (T a[], int n)
{//把数组a[0:n-1]中最大的元素通过冒泡移到右边
    for (int i = 0; i < n-1; i++)
        if (a[i] > a[i+1]) Swap(a[i], a[i+1]);
}
```

---

由于函数 Bubble 可以把最大的元素移到最右边, 因此可以用它来替换 SelectionSort 中的 Max, 从而得到一个新的排序函数 (见程序 2-9)。在新函数中, 元素比较的次数为  $(n-1)n/2$ , 与函数 SelectionSort 相同。

程序2-9 冒泡排序

---

```
template <class T>
void BubbleSort (T a[], int n)
{//对数组a[0:n-1]中的n个元素进行冒泡排序
    for (int i = n; i > 1; i--)
        Bubble(a,i);
}
```

---

### 最好、最坏和平均操作数

到目前为止, 在给出的例子中所使用的实例特征都很简单 (如输入数和 / 或输出数), 操作数都是这些特征的良性函数 (nice function)。如果把其他的一些操作也计算在内, 其中有些例子就可能变得很复杂了。例如, 由 Bubble(见程序2-8)所执行的交换次数不仅依赖于问题  $n$ , 而且依赖于数组  $a$  中的具体值。交换次数可在 0 到  $n-1$  之间变化。由于操作数不总是由所选择的实例特征唯一确定, 那么我们可能会问, 最好的、最坏的和平均的操作数分别是多少。下面就来讨论这个问题。

令  $P$  是一个程序。假定把操作数  $O_p(n_1, n_2, \dots, n_k)$  视为实例特征  $n_1, n_2, \dots, n_k$  的函数。对于任意程序实例  $I$ , 令  $operation_p(I)$  为该实例的操作数, 令  $S(n_1, n_2, \dots, n_k)$  为集合  $\{I | I \text{ 具有特征 } n_1, n_2, \dots, n_k\}$ 。则  $P$  最好的操作数为:

$$O_p^{BC}(n_1, n_2, \dots, n_k) = \min\{operation_p(I) \mid I \in S(n_1, n_2, \dots, n_k)\}$$

$P$  最坏的操作数为:

$$O_p^{WC}(n_1, n_2, \dots, n_k) = \max\{operation_p(I) \mid I \in S(n_1, n_2, \dots, n_k)\}$$

$P$  平均或期望的操作数为:

$$O_p^{AVG}(n_1, n_2, \dots, n_k) = \frac{1}{|S(n_1, n_2, \dots, n_k)|} \sum_{I \in S(n_1, \dots, n_k)} operation_p(I)$$

公式 $O_p^{AVG}$ 假定所有的 $I$ 都是完全相似的实例，否则该公式必须修改为：

$$O_p^{AVG}(n_1, n_2, \dots, n_k) = \sum_{I \in S(n_1, n_2, \dots, n_k)} p(I) \text{operation}_p(I)$$

其中 $p(I)$ 是实例 $I$ 可以被成功解决的概率（=次数/ $|S(n_1, n_2, \dots, n_k)|$ ）。

确定平均操作数通常是十分困难的，因此，在下面的几个例子中，仅分析最好和最坏的操作数。

例2-13 [顺序搜索] 在执行程序2-1顺序搜索的代码期间，我们想知道 $x$ 与 $a$ 中元素之间的比较次数，一个很自然的实例特征就是 $n$ 。不幸的是，比较的次数不是由 $n$ 唯一确定的。例如，如果 $n=100$ 且 $x=a[0]$ ，那么仅需要执行一次操作；如果 $x$ 不等于 $a$ 中的任何一个元素，则需要执行100次比较。

当 $x$ 是 $a$ 中的一员时称查找是成功的，其他情况都称为不成功查找。每当进行一次不成功查找，就需要执行 $n$ 次比较。对于成功查找来说，最好的比较次数是1，最坏的比较次数为 $n$ 。为了计算平均查找次数，假定所有的数组元素都是不同的，并且每个元素被查找的概率是相同的。成功查找的平均比较次数如下：

$$\frac{1}{n} \sum_{i=1}^n i = (n+1)/2$$

例2-14 [向有序数组中插入元素] 程序2-10向一个有序数组 $a[0:n-1]$ 中插入一个元素。 $a$ 中的元素在执行插入之前和插入之后都是按递增顺序排列的。例如，如果向数组 $a[0:5] = [1, 2, 6, 8, 9, 11]$ 中插入4，所得到的结果为 $a[0:6] = [1, 2, 4, 6, 8, 9, 11]$ 。当我们为新元素找到欲插入的位置时，必须把该位置右边的所有元素分别向右移动一个位置。对于本例，需要移动11, 9, 8和6，并把4插入到新空出来的位置 $a[2]$ 中。

程序2-10 向一个有序数组中插入元素

```
template<class T>
void Insert(T a[], int& n, const T& x)
// 向有序数组 a[0:n-1]中插入元素x
// 假定a的大小超过 n
int i;
for (i = n - 1; i >= 0 && x < a[i]; i--)
    a[i+1] = a[i];
a[i+1] = x;
n++; // 添加了一个元素
}
```

现在我们想知道执行程序2-10时， $x$ 与 $a$ 中元素之间的比较次数。一个很自然的实例特征就是初始数组 $a$ 的大小 $n$ 。最好或最少的比较次数为1，这种情况发生在 $x$ 被插入到数组尾部的时候。最大的比较次数为 $n$ ，这发生在 $x$ 被插入到 $a$ 的首部之时。为了估算平均比较次数，假定 $x$ 有相等的机会被插入到任一个可能的位置上（共有 $n+1$ 个可能的插入位置）。如果 $x$ 最终被插入到 $a$ 的 $i+1$ 处， $i \geq 0$ ，则执行的比较次数为 $n-i$ 。如果 $x$ 被插入到 $a[0]$ ，则比较次数为 $n$ 。所以平均比较次数为：

$$\frac{1}{n+1} \left( \sum_{i=0}^{n-1} (n-i) + n \right) = \frac{1}{n+1} \left( \sum_{j=1}^n j + n \right) = \frac{1}{n+1} (n(n+1)/2 + n) = n/2 + n/(n+1)$$

例2-15 [再看按名次排序] 假定已经使用函数Rank(见例2-9中的程序2-5)计算出一个数组中每个元素的名次，可以在原地把该数组中的元素按序重排，方法是从  $a[0]$  开始，每次检查一个元素。如果当前正在检查的元素为  $a[i]$  且  $r[i]=i$ ，那么可以跳过该元素，继续检查下一个元素；如果  $r[i] \neq i$ ，可以把  $a[i]$  与  $a[r[i]]$  进行交换，交换的结果是把原  $a[i]$  中的元素放到正确的排序位置 ( $r[i]$ ) 上去。这种交换操作在位置  $i$  处重复下去，直到应该排在位置  $i$  处的元素被交换到位置  $i$  处。之后，继续检查下一个位置。程序2-11给出了原地重排数组元素的函数Rearrange。

程序2-11 原地重排数组元素

```
template<class T>
void Rearrange(T a[], int n, int r[])
{
    // 原地重排数组元素
    for (int i = 0; i < n; i++)
        // 获取应该排在 a[i] 处的元素
        while (r[i] != i) {
            int t = r[i];
            Swap(a[i], a[t]);
            Swap(r[i], r[t]);
        }
}
```

程序2-11执行的最少交换次数为0(初始数组已经是按序排列)，最大的交换次数为  $2(n-1)$ 。注意每次交换操作至少把一个元素移到正确位置(如  $a[i]$ )，所以在  $n-1$  次交换之后，所有的  $n$  个元素已全部按序排列。因此，在最好的情况下，交换次数为 0，最坏情况下为  $2(n-1)$ (包括名次交换)。当使用本函数来代替程序2-6中的函数时，最坏情况下所需要的执行时间将增加，因为需要移动更多的元素(每次交换需要移动三次)，不过程序所需要的内存减少了。

例2-16 [再看选择排序] 程序2-7中选择排序函数的一个缺点是：即使元素已经按序排列，程序仍然继续运行。例如，即使在第二次循环过后数组元素可能已经按序排列，for循环仍需要执行  $n-1$  次。为了终止不必要的循环，在查找最大元素期间，可以顺便检查数组是否已按序排列。程序2-12给出了一个按照这种思想实现的选择排序函数。在该函数中，把查找最大元素的循环直接与函数SelectionSort合并在一起，而不是把它作为一个独立的函数。

程序2-12 及时终止的选择排序

```
template<class T>
void SelectionSort(T a[], int n)
{
    // 及时终止的选择排序
    bool sorted = false;
    for (int size = n; !sorted && (size > 1); size--) {
        int pos = 0;
        sorted = true;
        // 找最大元素
        for (int i = 1; i < size; i++)
            if (a[pos] <= a[i]) pos = i;
        else sorted = false; // 未按序排列
        Swap(a[pos], a[size - 1]);
    }
}
```

```
}  
}
```

对于程序 2-12 中的函数，其最好的情况出现在数组 *a* 最初已是有序数组的情形，此时，外部 *for* 循环仅执行一次，*a* 中元素之间的比较次数为  $n-1$ 。在最坏的情况下，外部 *for* 循环一直循环到 *size*=1，执行的比较次数为  $(n-1)n/2$ 。最好和最坏情况下的交换次数与程序 2-7 完全相同。注意在最坏的情况下，程序 2-12 可能要略微慢一些，因为它必须进行变量维护的额外工作。

例 2-17 [再看冒泡排序] 与选择排序的情形一样，可以重新设计一个及时终止的冒泡排序函数。如果在一次冒泡过程中没有发生元素互换，则说明数组已经按序排列，没有必要再继续进行冒泡过程。程序 2-13 给出了一个及时终止的冒泡排序函数。在最坏情况下所执行的比较次数与原来的函数（见程序 2-9）一样。在最好情况下比较次数为  $n-1$ 。

程序 2-13 及时终止的冒泡排序

```
template<class T>  
bool Bubble(T a[], int n)  
{//把 a[0:n-1] 中最大元素冒泡至右端  
    bool swapped = false; // 尚未发生交换  
    for (int i = 0; i < n - 1; i++)  
        if (a[i] > a[i+1]) {  
            Swap(a[i], a[i + 1]);  
            swapped = true; // 发生了交换  
        }  
    return swapped;  
}  
template<class T>  
void BubbleSort(T a[], int n)  
{// 及时终止的冒泡排序  
    for (int i = n; i > 1 && Bubble(a, i); i--);  
}
```

例 2-18 [插入排序] 程序 2-10 可以作为一个排序函数的基础。因为只有一个元素的数组是一个有序数组，所以可以从仅包含欲排序的  $n$  个元素的第一个元素的数组开始。通过把第二个元素插入到这个单元数组中，可以得到一个大小为 2 的有序数组。插入第三个元素可以得到一个大小为 3 的有序数组。按照这种方法继续进行下去，最终将得到一个大小为  $n$  的有序数组，这种排序方式为插入排序（insertion sort），函数 *InsertionSort*（见程序 2-14）正是按照这种思想实现的。为此，重写了函数 *Insert*（见程序 2-10），因为它执行了一些不必要的操作。实际上，还可以把 *Insert* 的代码直接嵌入到函数 *InsertionSort* 之中，从而得到另外一个插入排序函数（见程序 2-15）。等价地，也可以把函数 *Insert* 作为一个内联（inline）函数。注意，如果用代码 *Insert(a, i, a[i])* 取代 *InsertionSort* 函数中的 *for* 循环体，则程序将无法运行，因为 *Insert* 的形式参数是一个引用参数。

程序 2-14 插入排序

```
template<class T>  
void Insert(T a[], int n, const T& x)
```

```
{// 向有序数组 a[0:n-1]中插入元素x
int i;
for (i = n - 1; i >= 0 && x < a[i]; i--)
    a[i+1] = a[i];
a[i+1] = x;
}
template<class T>
void InsertionSort(T a[], int n)
{// 对 a[0:n-1]进行排序
    for (int i = 1; i < n; i++) {
        T t = a[i];
        Insert(a, i, t);
    }
}
```

程序2-15 另外一种插入排序

```
template<class T>
void InsertionSort(T a[], int n)
{
    for (int i = 1; i < n; i++) {
        //将a[i]插入a[0:i-1]
        T t = a[i];
        int j;
        for (j = i - 1; j >= 0 && t < a[j]; j--)
            a[j+1] = a[j];
        a[j+1] = t;
    }
}
```

程序2-14和程序2-15所执行的比较次数完全相同。在最好的情况下比较次数为  $n-1$ ,而在最坏的情况下比较次数为  $(n-1)n/2$ 。

### 2.3.3 执行步数

从上一节关于操作计数的例子中可以看出,利用操作计数方法来估算程序的时间复杂性忽略了所选择操作之外其他操作的开销。在统计执行步数 (step-count) 的方法中,要统计程序/函数中所有部分的时间开销。与操作计数一样,执行步数也是实例特征的函数。尽管任一个特定的程序可能会有若干个特征 (如输入个数,输出个数,输入和输出的大小),但可以把执行步数看成是其中一部分特征的函数。通常选择一些感兴趣的特征,例如,如果要了解程序的运行时间 (即时间复杂性) 是如何随着输入个数的增加而增加的,在这种情况下,可以把执行步数仅看成是输入个数的函数。因此,在确定一个程序的执行步数之前,必须确切地知道将要采用的实例特征。这些特征不仅定义了执行步数表达式中的变量,而且定义了以多少次计算作为一步。

选择了相关的实例特征以后,可以定义一个操作步 (step)。操作步是独立于所选特征的任意计算单位,10次加法可以视为一步,100次乘法也可以视为一步,但  $n$  次加法不能视为一步,其中  $n$  为实例特征。 $m/2$ 次加法或  $p+q$  次减法也都不能看成一步,其中  $m$ ,  $p$  和  $q$  都是实例特征。

定义 [程序步] 程序步 (program step) 可以定义为一个语法或语义意义上的程序片段, 该片段的执行时间独立于实例特征。

由一个程序步所表示的计算量可能与其他形式表示的计算量不同。例如, 下面这条完整语句:

```
return a + b + b*c + (a + b - c) / (a + b) + 4;
```

可以被视为一个程序步, 只要它的执行时间独立于所选用的实例特征。也可以把如下语句视为一个程序步:

```
x = y;
```

可以通过创建一个全局变量 count (其初值为0)来确定一个程序或函数为完成其预定任务所需要的执行步数。可以把 count 引入到程序语句之中, 每当原始程序或函数中的一条语句被执行时, 就为 count 累加上该语句所需要的执行步数。当程序或函数运行结束时所得到的 count 的值即为所需要的执行步数。

例2-19 把计算count 的语句引入到程序1-8之中, 可以得到程序2-16。在程序2-16运行结束时所得到的count 的值即为程序1-8的执行步数。

程序2-16 统计程序1-8的执行步数

---

```
template<class T>
T Sum(T a[], int n)
{// 计算 a[0:n - 1]中元素之和
    T tsum = 0;
    count++; // 对应于tsum = 0
    for (int i = 0; i < n; i++) {
        count++; // 对应于for语句
        tsum += a[i];
        count++; // 对应于赋值语句
    }
    count++; // 对应于最后一个for语句
    count++; //对应于return语句
    return tsum;
}
```

---

程序2-17作为程序2-16的一个简化版本, 仅仅用来确定 count值的改变。对于count的每一个初始值, 程序2-16和程序2-17最终所得到的count值都一样。在程序2-17的for循环中, count 的值被增加了2n。如果count的初值为0, 则在程序结束时它的值将变成2n+3。因此Sum (见程序1-8) 的每次调用需要执行2n+3步。

程序2-17 程序2-16的简化版本

---

```
template<class T>
T Sum(T a[], int n)
{//计算 a[0:n - 1]中元素之和
    for (int i = 0; i < n; i++)
        count += 2;
    count += 3;
```

---



```
return 0;
}
```

例2-20 把计算count的语句引入到程序1-9之中，可以得到程序2-18。

令 $t_{Rsum}(n)$ 为程序2-18结束时count所增加的值，可以看出 $t_{Rsum}(0)=2$ 。当 $n>0$ 时，count所增加的值为2加上调用函数Rsum（从then语句中）所增加的值。从 $t_{Rsum}(n)$ 的定义可知，额外的增值为 $t_{Rsum}(n-1)$ 。所以，如果count的初值为0，在程序结束时它的值将变成 $2+t_{Rsum}(n-1)$ ，其中 $n>0$ 。

程序2-18 统计程序1-9的执行步数

```
template<class T>
T Rsum(T a[], int n)
{//计算 a[0:n - 1]中元素之和
    count++; // 对应于if 条件
    if (n > 0) {count++; // 对应于return 和 Rsum 调用
        return Rsum(a, n - 1) + a[n-1];}
    count++; //对应于return
    return 0;
}
```

在分析一个递归程序的执行步数时，通常可以得到一个计算执行步数的递归等式（如 $t_{Rsum}(n) = 2 + t_{Rsum}(n-1)$ ， $n > 0$ 且 $t_{Rsum}(2) = 0$ ）。这种递归公式被称为递归等式（recurrence equation），或简称为递归。可以采用重复迭代的方法来计算递归等式，如：

$$t_{Rsum}(n) = 2 + t_{Rsum}(n-1) = 2 + 2 + t_{Rsum}(n-2) = 4 + t_{Rsum}(n-2) \dots = 2n + t_{Rsum}(0) = 2(n+1)$$

其中 $n \geq 0$ ，因此函数Rsum（见程序1-9）的执行步数为 $2(n+1)$ 。

比较程序1-8和程序1-9的执行步数，可以看到程序1-9的执行步数小于程序1-8的执行步数。不过不能因此断定程序1-8就比程序1-9慢，因为程序步不代表精确的时间单位。Rsum中的一步可能要比Sum中的一步花更多的时间，所以Rsum有可能要比Sum慢（预计可能会这样）。

执行步数可用来帮助我们了解程序的执行时间是如何随着实例特征的变化而变化的。从Sum的执行步数中可以看到，如果 $n$ 被加倍，程序运行时间也将加倍（近似地）；如果 $n$ 增加10倍，运行时间也会增加10倍。所以，可以预计，运行时间随着 $n$ 线性增长。我们称Sum是一个线性程序（其时间复杂性与实例特征 $n$ 呈线性关系）。

例2-21 [矩阵加] 考察程序2-19，它把存储在二维数组 $a[0:rows-1][0:cols-1]$ 和 $b[0:rows-1][0:cols-1]$ 中的两个矩阵相加。

程序2-19 矩阵加法

```
template<class T>
void Add( T **a, T **b, T **c, int rows, int cols)
{// 矩阵 a 和 b 相加得到矩阵 c.
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            c[i][j] = a[i][j] + b[i][j];
}
```

在程序2-19中引入count计数语句可得到程序2-20。程序2-21是程序2-20的一个简化版本，它计算同样的count值。检查程序2-21可以发现，如果count的初值为0，则在程序2-21结束时所得到的值为 $2rows*cols+2rows+1$ 。

从上述分析中可以看出，如果 $rows > cols$ ，最好交换程序2-19中的两条for语句，这样执行步数将变成 $2rows*cols+2cols+1$ 。注意，在本例中所使用的实例特征为rows和cols。

如果不想使用count计值语句，可以建立一张表，在该表中列出每条语句所需要的执行步数。建立这张表时，可以首先确定每条语句每一次执行所需要的步数以及该语句总的执行次数（即频率），然后利用这两个量就可以得到每条语句总的执行步数，把所有语句的执行步数加在一起即可得到整个程序的执行步数。

程序2-20 统计程序2-19的执行步数

```
template<class T>
void Add( T **a, T **b, T **c, int rows, int cols)
{//矩阵 a 和 b 相加得到矩阵 c.
    for (int i = 0; i < rows; i++) {
        count++; //对应于上一条for语句
        for (int j = 0; j < cols; j++) {
            count++; // 对应于上一条for语句
            c[i][j] = a[i][j] + b[i][j];
            count++; // 对应于赋值语句
        }
        count++; // 对应j 的最后一次for循环
    }
    count++; //对应i 的最后一次for循环
}
```

程序2-21 程序2-20的简化版本

```
template<class T>
void Add( T **a, T **b, T **c, int rows, int cols)
{//矩阵 a 和 b 相加得到矩阵 c.
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            c[i][j] = a[i][j] + b[i][j];
            count += 2;
        }
        count += 2;
    }
    count++;
}
```

一条语句的程序步数与该语句每次执行所需要的步数（s/e）之间有重要的区别，区别在于程序步数不能反映该语句的复杂程度。例如语句：

$x = \text{Sum}(a, m);$

的程序步数为1，而该语句的执行所导致的count值的实际变化为1加上调用Sum所导致的count值的变化（比如 $2m+3$ ）之和，因此该语句每次执行所需要的步数为 $1+2m+3=2m+4$ 。语句每次

执行所需要的步数通常记为  $s/e$ ，一条语句的  $s/e$  就等于执行该语句所产生的 count 值的变化量。

图2-4列出了函数 Sum(见程序1-8)中每条语句的执行频率及每次执行所需要的步数 ( $s/e$ )。该程序所需要的总的执行步数为  $2n+3$ 。注意 for 语句的频率为  $n+1$  而不是  $n$ ，因为  $i$  必须递增至  $n$ ，for 语句才能结束。

语 句	s/e	频 率	总 步 数
T Sum(T a[], int n)	0	0	0
{	0	0	0
T tsum = 0;	1	1	1
for(int i=0; i<n; i++)	1	$n+1$	$n+1$
tsum += a[i];	1	$n$	$n$
return tsum;	1	1	1
}	0	0	0
总 计			$2n+3$

图2-4 程序1-8的执行步数

图2-5采用  $s/e$  或列表方法列出了函数 Rsum(见程序1-9)的执行步数，而图2-6则列出了函数 Add(程序2-19)的执行步数。

语 句	s/e	频 率	总 步 数
T Rsum(T a[], int n)	0	0	0
{	0	0	0
if(n > 0)	1	$n+1$	$n+1$
return Rsum(a, n-1) + a[n-1];	1	$n$	$n$
return 0;	1	1	1
}	0	0	0
总 计			$2n+2$

图2-5 程序1-9的执行步数

语 句	s/e	频 率	总 步 数
void Add( T **a, ...)	0	0	0
{	0	0	0
for (int i = 0; i < rows; i++)	1	$rows+1$	$rows+1$
for (int j = 0; j < cols; j++)	1	$rows*(cols+1)$	$rows*cols+rows$
c[i][j] = a[i][j] + b[i][j];	1	$rows*cols$	$rows*cols$
}	0	0	0
总 计			$2rows*cols+2rows+1$

图2-6 程序2-19的执行步数

程序2-22用来转置一个  $rows \times rows$  的矩阵  $a[0:rows-1][0:rows-1]$ 。b 是 a 的转置，当且仅当对于所有的  $i$  和  $j$ ，有  $b[i][j]=a[j][i]$ 。

程序2-22 矩阵转置

```

template<class T>
void Transpose(T **a, int rows)
{// 对矩阵 a[0:rows-1][0:rows-1]进行转置
    for (int i = 0; i < rows; i++)
        for (int j = i+1; j < rows; j++)
            Swap(a[i][j], a[j][i]);
}

```

图2-7给出了一个执行步数表。让我们来推导第二条 for 循环语句的频率。对于 i 的每个值，该语句执行 rows-i 次，所以其频率为：

$$\sum_{i=0}^{rows-1} (rows-i) = \sum_{i=1}^{rows} i = rows(rows+1)/2$$

Swap 的频率为：

$$\sum_{i=0}^{rows-1} (rows-i-1) = \sum_{i=0}^{rows-1} i = rows(rows-1)/2$$

语 句	s/e	频 率	总 步 数
void Transpose(T **a, int rows)	0	0	0
{	0	0	0
for (int i = 0; i < rows; i++)	1	rows+1	rows+1
for (int j = i+1; j < rows; j++)	1	rows*(rows+1)/2	rows*(rows+1)/2
Swap(a[i][j], a[j][i]);	1	rows*(rows-1)/2	rows*(rows-1)/2
}	0	0	0
总 计			rows <sup>2</sup> +rows+1

图2-7 程序2-22的执行步数

在某些情况下，一条语句每次执行所需要的执行步数可能随时发生变化。例如，对于函数 Inef（见程序2-23）的赋值语句。函数 Inef 用非常低效的方式来计算数组元素的和：

$$\sum_{i=0}^j a[i] \text{ 对于 } j = 0, 1, \dots, n-1$$

程序2-23 低效的前缀求和程序

```

template <class T>
void Inef(T a[], T b[], int n)
{// 计算前缀和
    for (int j = 0; j < n; j++)
        b[j] = Sum(a, j + 1);
}

```

以上已经得出函数 Sum(a,n) 的执行步数为 2n+3，函数 Inef 中的赋值语句每执行一次所需要的步数为 2j+6。在这两个结果中分别加上了 1 个执行步。函数 Inef 中赋值语句的频率为 n，但该

语句总的执行步数并不是  $(2j+6)n$ ，而是：

$$\sum_{j=0}^{n-1} (2j+6) = n(n+5)$$

图2-8给出了该函数的完整分析。

语 句	s/e	频 率	总 步 数
void lnef(T a[], T b[], int n)	0	0	0
{	0	0	0
for (int j = 0; j < n; j++)	1	n+1	n+1
b[j] = Sum(a, j + 1);	2j+6	n	n(n+5)
}	0	0	0
总 计			$n^2+6n+1$

图2-8 程序2-23的执行步数

最好、最坏和平均操作数的概念可以很容易地扩充到执行步数应用中。下面的例子就阐述了这些概念。

例2-22 [顺序搜索] 图2-9和图2-10分别给出了函数SequentialSearch（见程序2-1）在最好和最坏情况下的执行步数分析。

语 句	s/e	频 率	总 步 数
int SequentialSearch(T a[], T& x, int n)	0	0	0
{	0	0	0
int i;	1	1	1
for (int i = 0; i < n && a[i] != x; i++)	1	1	1
if (i == n) return -1;	1	1	1
return i;	1	1	1
}	0	0	0
总 计			4

图2-9 最好情况下程序2-1的执行步数

语 句	s/e	频 率	总 步 数
int SequentialSearch(T a[], T& x, int n)	0	0	0
{	0	0	0
int i;	1	1	1
for (i = 0; i < n && a[i] != x; i++)	1	n+1	n+1
if (i == n) return -1;	1	1	1
return i;	1	0	0
}	0	0	0
总 计			n+3

图2-10 最坏情况下程序2-1的执行步数

为了分析一个成功查找的执行步数，假定数组  $a$  中的  $n$  个值都互不相同并且在一个成功的查找过程中， $x$  与数组中任何元素相匹配的概率都是一样的。在这样的假设下，一个成功查找的平均执行步数为  $n$  个可能的成功查找的执行步数之和除以  $n$ 。为了得到这个平均值，首先来分析一下当  $x=a[j]$  时的执行步数（如图 2-11 所示），其中  $j$  介于  $[0, n-1]$  范围内。

现在可以计算出成功查找的平均执行步数为：

$$t_{\text{SequentialSearch}}^{\text{AVG}}(n) = \frac{1}{n} \sum_{j=0}^{n-1} (j + 4) = (n + 7)/2$$

这个值稍小于一个不成功查找的执行步数的一半。

语 句	s/e	频 率	总 步 数
int SequentialSearch(T a[], T& x, int n)	0	0	0
{	0	0	0
int i;	1	1	1
for (i = 0; i < n && a[i] != x; i++)	1	j+1	j+1
if (i == n) return -1;	1	1	1
return i;	1	1	1
}	0	0	0
总 计			j+4

图2-11 当  $x=a[j]$  时程序 2-1 的执行步数

现在假设成功查找出现的概率为 80%，并且每个  $a[i]$  被查找的机会仍然相同，则 SequentialSearch 的平均执行步数为：

$$\begin{aligned}
 & 0.8 * (\text{成功查找的平均步数}) + 0.2 * (\text{不成功查找的执行步数}) \\
 &= 0.8(n+7)/2 + 0.2(n+3) \\
 &= 0.6n + 3.4
 \end{aligned}$$

例2-23 [向有序数组中插入元素] 函数 Insert 的最好和最坏执行步数分别如图 2-12 和图 2-13 所示。

语 句	s/e	频 率	总 步 数
void Insert(T a[], int& n, const T& x)	0	0	0
{	0	0	0
for (int i = n-1; i >= 0 && x < a[i]; i--)	1	1	1
a[i+1] = a[i];	0	0	0
a[i+1] = x;	1	1	1
n++;	1	1	1
}	0	0	0
总 计			3

图2-12 在最好情况下程序 2-10 的执行步数

语 句	s/e	频 率	总 步 数
void Insert(T a[], int& n, const T& x)	0	0	0
{	0	0	0
for (int i = n-1; i >= 0 && x < a[i]; i--)	1	n+1	n+1
a[i+1] = a[i];	1	n	n
a[i+1] = x;	1	1	1
n++;	1	1	1
}	0	0	0
总 计			2n+3

图2-13 在最坏情况下程序2-10的执行步数

为了计算平均执行步数，假定x被插入到任何位置上的概率是一样的（注：共有n+1个可能的位置）。如果x最终被插入到位置j处，j=0，则执行步数为2n-2j+3。所以平均执行步数为：

$$\frac{1}{n+1} \left( \sum_{j=0}^n (2n-2j+3) \right) = \frac{1}{n+1} \left[ 2 \sum_{k=0}^n k + 3(n+1) \right] = n+3$$

## 练习

- 试给出可能影响一个程序时间复杂性的其他因素。
- 在函数Sum（见程序1-8）的for循环中会执行多少次加法？
- 函数Factorial（见程序1-7）会执行多少次乘法？
- 修改程序2-6,把第一个for循环后面的代码替换为：释放a所占用的空间并把a更新到u中去。此外，把参数T a[]变成T\* &a。为了使新的代码能够运行，对应于a的实际参数必须是一个动态分配的数组。试测试新的代码。
- 创建一个输入数组a，使得函数Rearrange（见程序2-11）执行n-1次元素交换和n-1次行交换。
- 函数Add（见程序2-19）中矩阵元素对之间共执行了多少次加法？
- 函数Transpose（见程序2-22）共执行了多少次Swap操作？
- 试确定函数Mult（见程序2-24）共执行了多少次乘法，该函数实现两个n×n矩阵的乘法。

程序2-24 两个n×n矩阵的乘法

```
template<class T>
void Mult(T **a, T **b, T **c, int n)
{// 两个 n x n 矩阵 a 和 b 相乘得到 c.
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            T sum = 0;
            for (int k = 0; k < n; k++)
                sum += a[i][k] * b[k][j];
            c[i][j] = sum;
        }
}
```



16. 试确定函数Mult (见程序2-25) 共执行了多少次乘法, 该函数实现一个  $m \times n$  矩阵与一个  $n \times p$  矩阵之间的乘法。

程序2-25 一个  $m \times n$  矩阵与一个  $n \times p$  矩阵之间的乘法

```
template<class T>
void Mult(T **a, T **b, T **c, int m, int n, int p)
{// m x n 矩阵 a 与 n x p 矩阵 b相乘得到c
    for (int i = 0; i < m; i++)
        for (int j = 0; j < p; j++) {
            T sum = 0;
            for (int k = 0; k < n; k++)
                sum += a[i][k] * b[k][j];
            c[i][j] = sum;
        }
}
```

17. 试确定函数 Perm(见程序1-10)共执行了多少次Swap操作?

18. 函数MinMax(见程序2-26) 用来查找数组  $a[0:n-1]$  中的最大元素和最小元素。令  $n$  为实例特征。试问  $a$  中元素之间的比较次数是多少? 程序 2-27中给出了另一个查找最大和最小元素的函数。在最好和最坏情况下  $a$  中元素之间比较次数分别是多少? 试分析两个函数之间的相对性能。

程序2-26 查找最大和最小元素

```
template<class T>
bool MinMax(T a[], int n, int& Min, int& Max)
{// 寻找 a[0:n-1]中的最小元素和最大元素
// 如果数组中的元素数目小于1, 则返回 false
    if (n < 1) return false;
    Min = Max = 0; // 初始化
    for (int i = 1; i < n; i++) {
        if (a[Min] > a[i]) Min = i;
        if (a[Max] < a[i]) Max = i;
    }
    return true;
}
```

程序2-27 另一个查找最大和最小元素的函数

```
template<class T>
bool MinMax(T a[], int n, int& Min, int& Max)
{// 寻找 a[0:n-1]中的最小元素和最大元素
// 如果数组中的元素数目小于1, 则返回 false
    if (n < 1) return false;
    Min = Max = 0; // 初始化
    for (int i = 1; i < n; i++)
        if (a[Min] > a[i]) Min = i;
        else if (a[Max] < a[i]) Max = i;
```

```
    return true;
}
```

19. 递归函数 SequentialSearch(见程序2-2)中数组a与x 之间共执行了多少次比较？

20. 程序2-28给出了另外一个迭代式顺序搜索函数。在最坏情况下 x 与a中元素之间执行了多少次比较？把这个比较次数与程序 2-1中相应的比较次数进行比较，哪一个函数将运行得更快？为什么？

程序2-28 另一个顺序搜索函数

```
template<class T>
int SequentialSearch(T a[], const T& x, int n)
{// 在未排序的数组 a[0:n-1]中查找 x
// 如果找到，则返回相应位置，否则返回 - 1
    a[n] = x; // 假定该位置有效
    int i;
    for (i = 0; a[i] != x; i++);
    if (i == n) return - 1;
    return i;
}
```

21. 1) 在程序2-29中所有合适的位置插入count计值语句。

2) 通过删除原执行语句对 1) 中所得到的程序进行简化。简化后的程序所计算出的 count值应与1中程序所计算出的count值相同。

3) 当程序结束时count的值是多少？可以假定count的初值为0。

4) 采用频率方法分析程序2-29的执行步数，列出执行步数表。

程序2-29 练习21的函数

```
void D(int x[], int n)
{
    for (int i = 0; i < n; i += 2)
        x[i] += 2;
    int i = 1;
    while (i <= n/2) {
        x[i] += x[i+1];
        i++;
    }
}
```

22. 分别对如下函数完成练习 21：

1) Max(见程序1-31)。

2) MinMax(见程序2-26)。

3) MinMax(见程序2-27)，分析最坏情况下的执行步数。

4) Factorial(见程序1-7)。

5) PolyEval(见程序2-3)。

6) Horner(见程序2-4)。

7) Rank(见程序2-5)。

8) Perm(见程序1-10)。

9) SequentialSearch(见程序2-28)，分析最坏情况下的执行步数。

10) SelectionSort(见程序2-7)，分析最好和最坏情况下的执行步数。

11) SelectionSort(见程序2-12)，分析最好和最坏情况下的执行步数。

12) InsertionSort(见程序2-14)，分析最坏情况下的执行步数。

13) InsertionSort(见程序2-15)，分析最坏情况下的执行步数。

14) BubbleSort(见程序2-9)，分析最坏情况下的执行步数。

15) BubbleSort(见程序2-13)，分析最坏情况下的执行步数。

16) Mult(见程序2-24)。

23. 对如下函数完成练习21中的1、2和3:

1) Transpose(见程序2-22)

2) Inef(见程序2-23)

24. 计算如下函数的平均执行步数:

1) SequentialSearch(见程序2-2)

2) SequentialSearch(见程序2-28)

3) Insert(见程序2-10)

25. 1) 对于程序2-25完成练习21。

2) 在什么条件下适于交换最外层的两个for循环?

26. 试比较在最坏情况下，函数 SelectionSort(见程序2-12)、函数 InsertionSort(见程序2-15) 以及函数 Bubble Sort(见程序2-13) 中元素移动的次数。利用程序2-11完成按名次排序。请说明对于大型数组，这两种排序方法之间的相对性能。

27. 在最坏的情况下一个程序所需要的运行时间和内存都必须是最大的吗? 证明你的结论。

## 2.4 渐进符号 ( $O$ 、 $\Omega$ 、 $\Theta$ 、 $o$ )

之所以要确定程序的操作计数和执行步数有两个重要的原因: 1) 为了比较两个完成同一功能的程序的时间复杂性; 2) 为了预测随着实例特征的变化, 程序运行时间的变化量。操作计数和执行步数都不能够非常精确地描述时间复杂性。在使用操作计数时, 我们把注意力集中在某些“关键”的操作上, 而忽略了所有其他操作。执行步数方法则试图通过关注所有的操作以便克服操作计数方法的不足, 然而, “执行步”的概念本身就不精确。指令  $x=y$  和  $x=y+z+(x/y)$  都可以被称为一步。正由于执行步数的不精确性, 所以不便于用来进行比较。不过如果两个程序之间的执行步数相差非常大, 比如一个为  $3n+3$ , 一个为  $100n+10$ , 则另当别论。我们可以非常保险地预测, 一个执行步数为  $3n+3$  的程序将比执行步数为  $100n+10$  的程序需要更少的执行时间。但在这种情况下其实并不需要精确地知道执行步数为  $100n+10$ , 类似于“大概为  $80n$  或  $85n$  或  $75n$ ”就足以得到相同的结论。

如果有两个程序的时间复杂性分别为  $c_1 n^2 + c_2 n$  和  $c_3 n$ , 那么可以知道, 对于足够大的  $n$ , 复杂性为  $c_3 n$  的程序将比复杂性为  $c_1 n^2 + c_2 n$  的程序运行得快。对于比较小的  $n$  值, 两者都有可能成为较快的程序 (取决于  $c_1$ ,  $c_2$  和  $c_3$ )。如果  $c_1=1$ ,  $c_2=2$ ,  $c_3=100$ , 那么当  $n < 98$  时  $c_1 n^2 + c_2 n < c_3 n$ , 当  $n > 98$  时  $c_1 n^2 + c_2 n > c_3 n$ ; 如果  $c_1=1$ ,  $c_2=2$ ,  $c_3=1000$ , 那么当  $n < 998$  时  $c_1 n^2 + c_2 n < c_3 n$ 。因此, 对于大多数情况, 足以得出结论  $c_1 n^2 = t_p^{wc}(n) = c_2 n^2$  或  $t_q^{wc}(n, m) = c_1 n + c_2 m$ , 其中  $c_1$  和  $c_2$  为非负常数。

不管  $c_1$ 、 $c_2$  和  $c_3$  的值是多少, 总会存在一个门槛值, 使得当  $n$  小于这个门槛值时, 复杂性为

$c_3 n$  的程序要比复杂性为  $c_1 n^2 + c_2 n$  的程序运行得快。这个门槛值被称为均衡点 (breakeven point)。如果均衡点为 0, 那么复杂性为  $c_3 n$  的程序总是要比复杂性为  $c_1 n^2 + c_2 n$  的程序运行得快 (或者至少一样快)。精确的均衡点不可能通过分析来确定, 必须在计算机上实际运行程序才能确定, 所以确定  $c_1$ 、 $c_2$  和  $c_3$  的精确值并没有多大帮助。

前述讨论的目的是为了引入新的符号 (或记号), 利用新符号可以写出关于程序时间和空间复杂性的具体公式 (尽管不够精确)。这种符号被称为渐进符号 (asymptotic notation), 它可以描述大型实例特征下时间或空间复杂性的具体表现。在下面的讨论中  $f(n)$  表示一个程序的时间或空间复杂性, 它是实例特征  $n$  的函数。由于一个程序的时间和空间需求是一个非负值, 所以可以假定对于  $n$  的所有取值, 函数  $f$  的值非负。由于  $n$  表示一个实例特征, 所以可以进一步假定  $n \geq 0$ 。即将讨论的渐进符号允许我们对于足够大的  $n$  值, 给出  $f$  的上限值和/或下限值。

### 2.4.1 大写 O 符号

大写 O 符号给出了函数  $f$  的一个上限。

定义 [大写 O 符号]  $f(n) = O(g(n))$  当且仅当存在正的常数  $c$  和  $n_0$ , 使得对于所有的  $n \geq n_0$ , 有  $f(n) \leq c g(n)$ 。

上述定义表明, 函数  $f$  顶多是函数  $g$  的  $c$  倍, 除非  $n$  小于  $n_0$ 。因此对于足够大的  $n$  (如  $n \geq n_0$ ),  $g$  是  $f$  的一个上限 (不考虑常数因子  $c$ )。在为函数  $f$  提供一个上限函数  $g$  时, 通常使用比较简单的函数形式。比较典型的形式是含有  $n$  的单个项 (带一个常数系数)。图 2-14 列出了一些常用的  $g$  函数及其名称。对于图 2-14 中的对数函数  $\log n$ , 没有给出对数基, 原因是对于任何大于 1 的常数  $a$  和  $b$  都有  $\log_a n = \log_b n / \log_b a$ , 所以  $\log_a n$  和  $\log_b n$  都有一个相对的乘法系数  $1 / \log_b a$ , 其中  $a$  是一个常量。

函 数	名 称
1	常数
$\log n$	对数
$n$	线性
$n \log n$	$n$ 个 $\log n$
$n^2$	平方
$n^3$	立方
$2^n$	指数
$n!$	阶乘

图 2-14 常用的渐进函数

例 2-24 [线性函数] 考察  $f(n) = 3n + 2$ 。当  $n \geq 2$  时,  $3n + 2 \leq 3n + n = 4n$ , 所以  $f(n) = O(n)$ ,  $f(n)$  是一个线性变化的函数。采用其他方式也可以得到同样结论, 例如, 对于  $n > 0$ , 有  $3n + 2 \leq 10n$ , 可以通过选择  $c = 10$  以及  $n_0 > 0$  来满足大 O 定义。此外, 由于当  $n \geq 1$  时, 有  $3n + 2 \leq 3n + 2n = 5n$ , 所以通过选择  $c = 5$  以及  $n_0 = 1$  也可以满足大 O 定义。用来满足大 O 定义的  $c$  和  $n_0$  的值并不重要, 因为只需说明  $f(n) = O(g(n))$ , 公式中并未出现  $c$  和  $n_0$ 。

对于函数  $f(n) = 3n + 3$ , 当  $n \geq 3$  时, 有  $3n + 3 \leq 3n + n = 4n$ , 所以  $f(n) = O(n)$ 。类似地, 对于  $n$

$n_0=6$ , 有  $f(n)=100n+6$   $100n+n=101n$ , 因此,  $100n+6=O(n)$ 。正如所期望的那样,  $3n+2$ ,  $3n+3$  以及  $100n+6$  都满足  $n$  的大  $O$  定义, 也即它们都是线性函数 (对于一定的  $n$ )。

例2-25 [平方函数] 假定  $f(n) = 10n^2 + 4n + 2$ 。对于  $n \geq 2$ , 有  $f(n) \leq 10n^2 + 5n$ 。由于当  $n \geq 5$  时有  $5n \leq n^2$ , 因此对于  $n \geq n_0=5$ ,  $f(n) \leq 10n^2 + n^2 = 11n^2$ , 所以  $f(n) = O(n^2)$ 。

考察另外一个具有平方复杂性的例子  $f(n)=1000n^2 + 100n - 6$ 。可以很容易地看出对于所有  $n$ , 有  $f(n) \leq 1000n^2 + 100n$ 。此外, 对于  $n \geq 100$ , 有  $100n \leq n^2$ , 因此对于  $n \geq n_0=100$ , 有  $f(n) < 1001n^2$ , 因而  $f(n) = O(n^2)$ 。

例2-26 [指数函数] 考察一个具有指数复杂性的例子  $f(n) = 6 * 2^n + n^2$ 。可以观察到对于  $n \geq 4$ , 有  $n^2 \leq 2^n$ , 所以对于  $n \geq 4$ , 有  $f(n) \leq 6 * 2^n + 2^n = 7 * 2^n$ , 因此  $6 * 2^n + n^2 = O(2^n)$ 。

例2-27 [常数函数] 当  $f(n)$  是一个常数时, 比如  $f(n)=9$  或  $f(n)=2033$ , 可以记为  $f(n)=O(1)$ 。这种写法的正确性很容易证明。例如, 对于  $f(n)=9$   $9 \leq 1$ , 只要令  $c=9$  以及  $n_0=0$  即可得  $f(n)=O(1)$ 。同样地, 对于  $f(n)=2033$   $2033 \leq 1$ , 只要令  $c=2033$  以及  $n_0=0$  即可。

例2-28 [松散界限] 当  $n \geq 2$  时有  $3n+3 \leq 3n^2$ , 所以  $3n+3=O(n^2)$ , 虽然  $n^2$  是  $3n+3$  的一个上限, 但不是最小上限, 因为可以找到一个更小的函数 (在本例中为线性函数) 来满足大  $O$  定义。

当  $n \geq 2$  时有  $10n^2 + 4n + 2 \leq 10n^4$ , 所以  $10n^2 + 4n + 2 = O(n^4)$ , 但  $n^4$  同样不是  $10n^2 + 4n + 2$  的最小上限。

类似地,  $6n2^n + 20 = O(n^2 2^n)$ , 但  $n^2 2^n$  不是最小上限, 因为可以找到一个更小的上限为  $n2^n$ 。因此,  $6n2^n + 20 = O(n2^n)$ 。

注意在前面例子的中所使用的推导策略是: 用次数低的项目 (如  $n$ ) 替换次数高的项目 (如  $n^2$ ), 直到剩下一个单项为止。

例2-29 [错误界限]  $3n+2 = O(1)$ , 因为不存在  $c > 0$  及  $n_0$ , 使得对于所有的  $n \geq n_0$ , 有  $3n+2 < c$ 。可以使用反证法来证明这个结论。假定存在这样的  $c$  及  $n_0$ , 使得对于所有的  $n \geq n_0$ , 有  $3n+2 < c$ , 则可以得到  $n < (c-2)/3$ , 因此当  $n > \max\{n_0, (c-2)/3\}$  时, 将出现矛盾的结论。

为了证明  $10n^2 + 4n + 2 = O(n)$ , 首先假定  $10n^2 + 4n + 2 = O(n)$ , 因此, 总存在一个正数  $c$  和  $n_0$ , 使得对于所有的  $n \geq n_0$ , 有  $10n^2 + 4n + 2 \leq cn$ 。不等式两边同时除以  $n$  可得: 对于所有的  $n \geq n_0$ , 有  $10n + 4 + 2/n \leq c$ 。这个不等式不总是成立的, 因为不等式的左边随着  $n$  的增长而增大, 而不等式的右边则保持不变。比如取  $n > \max\{n_0, (c-4)/10\}$ , 将得到矛盾的结论。

$f(n) = 3n^2 2^n + 4n2^n + 8n^2 = O(2^n)$ 。为了证明这个不等式, 假定  $f(n) = O(2^n)$ , 因此, 总存在一个  $c > 0$  和  $n_0$ , 使得对于所有的  $n \geq n_0$ , 有  $f(n) \leq c * 2^n$ 。不等式两边同时除以  $2^n$  可得: 对于所有的  $n \geq n_0$ , 有  $3n^2 + 4n + 8n^2 / 2^n \leq c$ 。与上例一样, 这个不等式的左边随着  $n$  的增长而增大, 而不等式的右边则是一个常数, 因此对于一个“足够大”的  $n$ , 不等式将不成立。

如例2-28所示, 语句  $f(n) = O(g(n))$  仅表明对于所有的  $n \geq n_0$ ,  $cg(n)$  是  $f(n)$  的一个上限。它并未指出该上限是否为最小上限,  $n = O(n^2)$ ,  $n = O(n^{2.5})$ ,  $n = O(n^3)$  和  $n = O(2^n)$ 。为了使语句  $f(n) = O(g(n))$  有实际意义, 其中的  $g(n)$  应尽量地小。因此常见的是  $3n+3 = O(n)$ , 而不是  $3n+3 = O(n^2)$ , 尽管后者也是正确的。

注意  $f(n) = O(g(n))$  并不等价于  $O(g(n)) = f(n)$ 。实际上,  $O(g(n)) = f(n)$  是无意义的。在这里, 使用符号  $=$  是不确切的, 因为  $=$  通常表示相等关系。可以通过把符号  $=$  读作“是”而不是“等于”来避免这种矛盾。

定理2-1给出了一个非常有用的结论，利用该结论可以获取 $f(n)$ 的序（即 $f(n)=O(g(n))$ 中的 $g(n)$ ），这里， $f(n)$ 是一个关于 $n$ 的多项式。

定理2-1 如果 $f(n)=a_m n^m + \dots + a_1 n + a_0$ 且 $a_m > 0$ ，则 $f(n)=O(n^m)$ 。

证明 对于所有的 $n$  1有：

$$\begin{aligned} f(n) &\leq \sum_{i=0}^m |a_i| n^i \\ &\leq n^m \sum_{i=0}^m |a_i| n^{i-m} \\ &\leq n^m \sum_{i=0}^m |a_i| \end{aligned}$$

所以 $f(n)=O(n^m)$ 。

例2-30 把定理2-1应用到例2-24、2-25和2-27中去。对于例2-24中的三个函数，都有 $m=1$ ，因此这三个函数的序均为 $O(n)$ 。对于例2-25中的函数， $m=2$ ，因此所有函数的序均为 $O(n^2)$ 。而对于例2-27中的两个常量来说， $m=0$ ，因此这两个常量的序均为 $O(1)$ 。

可以对例2-29中所采用的策略进行扩充，以证明一个上限尽管满足大 $O$ 定义，但它并不是真正需要的上限，这正是定理2-2所要做的事情。采用定理2-2通常要比采用大 $O$ 定义更容易证明 $f(n)=O(g(n))$ 。

定理2-2 [大 $O$ 比率定理] 对于函数 $f(n)$ 和 $g(n)$ ，若 $\lim_n f(n)/g(n)$ 存在，则 $f(n)=O(g(n))$ 当且仅当存在确定的常数 $c$ ，有 $\lim_n f(n)/g(n) \leq c$ 。

证明 如果 $f(n)=O(g(n))$ ，则存在 $c>0$ 及某个 $n_0$ ，使得对于所有的 $n \geq n_0$ ，有 $f(n)/g(n) \leq c$ ，因此 $\lim_n f(n)/g(n) \leq c$ 。接下来假定 $\lim_n f(n)/g(n) \leq c$ ，它表明存在一个 $n_0$ ，使得对于所有的 $n \geq n_0$ ，有 $f(n) \leq \max\{1, c\} * g(n)$ 。

例2-31 因为 $\lim_n (3n+2)/n = 3$ ，所以 $3n+2=O(n)$ ；因为 $\lim_n (10n^2+4n+2)/n^2 = 10$ ，所以 $10n^2+4n+2=O(n^2)$ ；因为 $\lim_n (6*2^n+2^n)/2^n = 6$ ，所以 $6*2^n+2^n=O(2^n)$ ；因为 $\lim_n (2n^2-3)/n^4 = 0$ ，所以 $2n^2-3=O(n^4)$ ；因为 $\lim_n (3n^2+5)/n = \infty$ ，所以 $3n^2+5 \neq O(n)$ 。

## 2.4.2 符号

符号与大 $O$ 符号类似，它用来估算函数 $f$ 的下限值。

定义 [符号]  $f(n) = \Omega(g(n))$  当且仅当存在正的常数 $c$ 和 $n_0$ ，使得对于所有的 $n \geq n_0$ ，有 $f(n) \geq cg(n)$ 。

使用定义 $f(n) = \Omega(g(n))$ 是为了表明，函数 $f$ 至少是函数 $g$ 的 $c$ 倍，除非 $n$ 小于 $n_0$ 。因此对于足够大的 $n$ （如 $n \geq n_0$ ）， $g$ 是 $f$ 的一个下限（不考虑常数因子 $c$ ）。与大 $O$ 定义的应用一样，通常仅使用单项形式的 $g$ 函数。

例2-32 对于所有的 $n$ ，有 $f(n)=3n+2>3n$ ，因此 $f(n)=\Omega(n)$ 。同样地， $f(n)=3n+3>3n$ ，所以有

$f(n) = (n)$ ;  $f(n) = 100n + 6 > 100n$ , 所以  $100n + 6 = (n)$ 。因而  $3n + 2$ ,  $3n + 3$  和  $100n + 6$  都是带有下界的线性函数。

对于所有的  $n \geq 0$ , 有  $f(n) = 10n^2 + 4n + 2 > 10n^2$ , 因此  $f(n) = (n^2)$ 。同样地,  $1000n^2 + 100n - 6 = (n^2)$ 。由于  $6 \cdot 2^n + n^2 > 6 \cdot 2^n$ , 所以  $6 \cdot 2^n + n^2 = (2^n)$ 。

同样也可以得到  $3n + 3 = (1)$ ,  $10n^2 + 4n + 2 = (n)$ ,  $10n^2 + 4n + 2 = (1)$ ,  $6 \cdot 2^n + n^2 = (n^{100})$ ,  $6 \cdot 2^n + n^2 = (n^{50.2})$ ,  $6 \cdot 2^n + n^2 = (n^2)$ ,  $6 \cdot 2^n + n^2 = (n)$  和  $6 \cdot 2^n + n^2 = (1)$ 。

为了说明  $3n + 2 = (n^2)$ , 可以先假定  $3n + 2 = (n^2)$ , 因此存在正数  $c$  和  $n_0$ , 使得对于所有的  $n \geq n_0$ , 有  $3n + 2 \leq cn^2$ , 所以对于所有  $n \geq n_0$ , 有  $cn^2 / (3n + 2) \geq 1$ , 这个不等式不可能总成立, 因为不等式的左边将会随着  $n$  的增大而变得无限大。

与大O定义中的情形一样, 可能存在若干个函数  $g(n)$  满足  $f(n) = (g(n))$ 。 $g(n)$  仅是  $f(n)$  的一个下限。为了使语句  $f(n) = (g(n))$  更有实际意义, 其中的  $g(n)$  应足够地大。因此常用的是  $3n + 3 = (n)$  及  $6 \cdot 2^n + n^2 = (2^n)$ , 而不是  $3n + 3 = (1)$  及  $6 \cdot 2^n + n^2 = (1)$ , 尽管后者也是正确的。

定理2-3是与定理2-1相类似的一个定理, 它是关于  $\Theta$  符号的定理。

定理2-3 如果  $f(n) = a_m n^m + \dots + a_1 n + a_0$  且  $a_m > 0$ , 则  $f(n) = (n^m)$ 。

证明 见练习31。

例2-33 根据定理2-3可知,  $3n + 2 = (n)$ ,  $10n^2 + 4n + 2 = (n^2)$ ,  $100n^4 + 3500n^2 + 82n + 8 = (n^4)$ 。

定理2-4是与定理2-2类似的一个定理, 采用定理2-4通常要比采用  $\Theta$  定义更容易证明  $f(n) = (g(n))$ 。

定理2-4 [比率定理] 对于函数  $f(n)$  和  $g(n)$ , 若  $\lim_n g(n) / f(n)$  存在, 则  $f(n) = (g(n))$  对于确定的常数  $c$ , 有  $\lim_n g(n) / f(n) = c$ 。

证明 见练习32。

例2-34 因为  $\lim_n (3n + 2) = 1/3$ , 所以  $3n + 2 = (n)$ ; 因为  $\lim_n n^2 / (10n^2 + 4n + 2) = 0.1$ , 所以  $10n^2 + 4n + 2 = (n^2)$ 。因为  $\lim_n 2^n / (6 \cdot 2^n + n^2) = 1/6$ , 所以  $6 \cdot 2^n + n^2 = (2^n)$ ; 因为  $\lim_n n / (6n^2 + 2) = 0$ , 所以  $6n^2 + 2 = (n)$ ; 因为  $\lim_n n^3 / (3n^2 + 5) = \infty$ , 所以  $3n^2 + 5 = (n^3)$ 。

#### 2.4.3 $\Theta$ 符号

$\Theta$  符号适用于同一个函数  $g$  既可以作为  $f$  的上限也可以作为  $f$  的下限的情形。

定义 [  $\Theta$  符号 ]  $f(n) = \Theta(g(n))$  当且仅当存在正常数  $c_1$ ,  $c_2$  和某个  $n_0$ , 使得对于所有的  $n \geq n_0$ , 有  $c_1 g(n) \leq f(n) \leq c_2 g(n)$ 。

使用定义  $f(n) = \Theta(g(n))$  是为了表明, 函数  $f$  介于函数  $g$  的  $c_1$  倍和  $c_2$  倍之间, 除非  $n$  小于  $n_0$ 。因此对于所有足够大的  $n$  (如  $n \geq n_0$ ),  $g$  既是  $f$  的上限也是  $f$  的下限 (不考虑常数因子  $c$ )。与大O定义和  $\Omega$  定义的应用一样, 通常仅使用单项形式的  $g$  函数。

例2-35 从例2-24, 2-25, 2-26和2-32中可以得到:  $3n + 2 = \Theta(n)$ ,  $3n + 3 = \Theta(n)$ ,  $100n + 6 = \Theta(n)$ ,  $10n^2 + 4n + 2 = \Theta(n^2)$ ,  $1000n^2 + 100n - 6 = \Theta(n^2)$ ,  $6 \cdot 2^n + n^2 = \Theta(2^n)$ 。

由于对于  $n \geq 16$ ,  $\log_2 n < 10 \cdot \log_2 n + 4 \leq 11 \cdot \log_2 n$ , 所以  $10 \cdot \log_2 n + 4 = \Theta(\log_2 n)$ 。前面曾指出  $\log_a n$  等于  $\log_b n$  乘以一个常数, 所以可以把  $\Theta(\log_a n)$  简写为  $\Theta(\log n)$ 。



在例2-29中证明了  $3n+2 = O(1)$ ，所以  $3n+2 = \Theta(1)$ 。类似地，可以得到  $3n+3 = \Theta(1)$ ， $100n+6 = \Theta(1)$ 。因为  $3n+3 = O(n^2)$ ，所以  $3n+3 = \Theta(n^2)$ ；因为  $10n^2+4n+2 = O(n)$ ，所以  $10n^2+4n+2 = \Theta(n)$ ；因为  $10n^2+4n+2 = O(1)$ ，所以  $10n^2+4n+2 = \Theta(1)$ 。

因为  $6 \cdot 2^n + n^2 = O(n^2)$ ，所以  $6 \cdot 2^n + n^2 = \Theta(n^2)$ 。同样道理， $6 \cdot 2^n + n^2 = \Theta(n^{100})$ ， $6 \cdot 2^n + n^2 = \Theta(1)$ 。

正如前面所提到的，在实际应用过程中，仅使用乘法系数为1的 $g$ 函数，所以几乎从来不会采用  $3n+3=O(3n)$ ，或  $10=O(100)$ ，或  $10n^2+4n+2=O(4n^2)$ ，或  $6 \cdot 2^n + n^2 = O(6 \cdot 2^n)$ ，或  $6 \cdot 2^n + n^2 = \Theta(4 \cdot 2^n)$ ，即使这些语句都是正确的。

**定理2-5** 如果  $f(n) = a_m n^m + \dots + a_1 n + a_0$  且  $a_m > 0$ ，则  $f(n) = \Theta(n^m)$ 。

**证明** 见练习31。

**例2-36** 根据定理2.5可知， $3n+2 = \Theta(n)$ ， $10n^2+4n+2 = \Theta(n^2)$ ， $100n^4+3500n^2+82n+8 = \Theta(n^4)$ 。

定理2-6是与定理2-2和定理2-4类似。

**定理2-6** [ $\Theta$ 比率定理] 对于函数  $f(n)$  和  $g(n)$ ，若  $\lim_n f(n)/g(n)$  及  $\lim_n g(n)/f(n)$  存在，则  $f(n) = \Theta(g(n))$  当且仅当存在确定的常数  $c$ ，有  $\lim_n f(n)/g(n) = c$  及  $\lim_n g(n)/f(n) = c$ 。

**证明** 见练习32。

**例2-37** 因为  $\lim_n (3n+2)/n = 3$  且  $\lim_n n/(3n+2) = 1/3 < 3$ ，所以  $3n+2 = \Theta(n)$ ；因为  $\lim_n (10n^2+4n+2)/n^2 = 10$  且  $\lim_n n^2/(10n^2+4n+2) = 0.1 < 10$ ，所以  $10n^2+4n+2 = \Theta(n^2)$ ；因为  $\lim_n (6 \cdot 2^n + n^2)/2^n = 6$  且  $\lim_n 2^n/(6 \cdot 2^n + n^2) = 1/6 < 6$ ，所以  $6 \cdot 2^n + n^2 = \Theta(2^n)$ ；因为  $\lim_n (6n^2+2)/n = \infty$ ，所以  $6n^2+2 = \Theta(n^2)$ 。

#### 2.4.4 小写o符号

**定义** [小写o]  $f(n) = o(g(n))$  当且仅当  $f(n) = O(g(n))$  且  $f(n) \neq \Theta(g(n))$ 。

**例2-38** [小写o] 因为  $3n+2 = O(n^2)$  且  $3n+2 \neq \Theta(n^2)$ ，所以  $3n+2 = o(n^2)$ ，但  $3n+2 \neq o(n)$ 。类似地， $10n^2+4n+2 = o(n^3)$ ，但  $10n^2+4n+2 \neq o(n^2)$ 。

#### 2.4.5 特性

下面的定理可用于有关渐进符号的计算。

**定理2-7** 对于任一个实数  $x > 0$  和任一个实数  $\varepsilon > 0$ ，下面的结论都是正确的：

- 1) 存在某个  $n_0$  使得对于任何  $n > n_0$ ，有  $(\log n)^x < (\log n)^{x+\varepsilon}$ 。
- 2) 存在某个  $n_0$  使得对于任何  $n > n_0$ ，有  $(\log n)^x < n$ 。
- 3) 存在某个  $n_0$  使得对于任何  $n > n_0$ ，有  $n^x < n^{x+\varepsilon}$ 。
- 4) 对于任意实数  $y$ ，存在某个  $n_0$  使得对于任何  $n > n_0$ ，有  $n^x (\log n)^y < n^{x+\varepsilon}$ 。
- 5) 存在某个  $n_0$  使得对于任何  $n > n_0$ ，有  $n^x < 2^n$ 。

**证明** 可参考各个函数的定义。

**例2-39** 根据定理2-7，可以得到如下结论： $n^3 + n^2 \log n = \Theta(n^3)$ ；对于任意自然数  $k$ ， $2^n/n^2 = \Theta(n^k)$ ； $n^4 + n^{2.5} \log^{20} n = \Theta(n^4)$ ； $2^n n^4 \log^3 n + 2^n n^4 / \log n = \Theta(2^n n^4 \log^3 n)$ 。

图2-15列出了一些常用的有关  $O$ ， $\Theta$  和  $o$  的标记，在该表中除  $n$  以外所有符号均为正常数。

图2-16给出了一些关于和与积的有用的引用规则。其中，+可以是 $O$ 、 $\Theta$ 之一。

图2-15和2-16为我们使用渐进符号来描述程序的时间复杂性（或执行步数）做好了准备。

$f(n)$	渐进符号
E1 $c$	$\Theta(1)$
E2 $\sum_{i=0}^k c_i n^i$	$\Theta(n^k)$
E3 $\sum_{i=1}^n i$	$\Theta(n^2)$
E4 $\sum_{i=1}^n i^2$	$\Theta(n^3)$
E5 $\sum_{i=1}^n i^k, k>0$	$\Theta(n^{k+1})$
E6 $\sum_{i=0}^n r^i, r>1$	$\Theta(r^n)$
E7 $n!$	$\Theta((n/e)^n)$
E8 $\sum_{i=1}^n 1/i$	$\Theta(\log n)$

图2-15 渐进标记

11	$\{f(n) = \Theta(g(n))\} \rightarrow \sum_{n=a}^v f(n) = \Theta(\sum_{n=a}^v g(n))$
12	$\{f_i(n) = \Theta(g_i(n)), 1 \leq i \leq k\} \rightarrow \sum_1^k f_i(n) = \Theta(\max_{1 \leq i \leq k} \{g_i(n)\})$
13	$\{f_i(n) = \Theta(g_i(n)), 1 \leq i \leq k\} \rightarrow \prod_1^k f_i(n) = \Theta(\prod_1^k g_i(n))$
14	$\{f_1(n) = O(g_1(n)), f_2(n) = \Theta(g_2(n))\} \rightarrow f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$
15	$\{f_1(n) = \Theta(g_1(n)), f_2(n) = \Omega(g_2(n))\} \rightarrow f_1(n) + f_2(n) = \Omega(g_1(n) + g_2(n))$
16	$\{f_1(n) = O(g(n)), f_2(n) = \Theta(g(n))\} \rightarrow f_1(n) + f_2(n) = \Theta(g(n))$

图2-16 关于 $O$ ， $\Theta$ 和 $\Omega$ 的引用规则

#### 2.4.6 复杂性分析举例

让我们重新检查一下2.3.3节所分析的时间复杂性。对于函数Sum(见程序1-8)，已经知道 $t_{Sum}(n) = 2n+3$ ，所以 $t_{Sum}(n) = \Theta(n)$ 。此外， $t_{Rsum}(n) = n(m+1)+2 = \Theta(mn)$ ， $t_{Add}(m, n) = 2mn + 2n+1 = \Theta(mn)$ ， $t_{Transpose}(n) = (n-1)(4n+2)/2 = \Theta(n^2)$ 。

我们已经知道 $t_{SequentialSearch}^{WC}(n) = n+3 = \Theta(n)$ 。由于 $n+3$ 是最坏情况下的复杂性，所以它是 $t_{SequentialSearch}(n)$ 的一个上限，因此 $t_{SequentialSearch}(n) = O(n)$ 。后面这个公式表明，存在正常数 $c$ 和 $n_0$ ，使得对于所有的 $n \geq n_0$ ，函数SequentialSearch的计算时间受限于 $cn$ 。注意 $t_{SequentialSearch}(n)$ 实际上是一个多值函数，因为对于不同的 $n$ ，它有不同的取值。此外，因为在最好的情况下， $x=a[0]$ ，此时，

$t_{\text{SequentialSearch}}(n)=4$ ，所以 $t_{\text{SequentialSearch}}(n)= (1)$ 。 $t_{\text{SequentialSearch}}^{\text{AVG}}(n)= (n+7)/2+(1- ) (n+3)=\Theta(n)$ ，其中  $x$  存在于数组a 中的概率。

尽管上一段中正确地使用了O、 $\Theta$ 和 $\Omega$ 符号，但我们还是无法进行精确的执行步数分析。实际上，如果不需要确定精确的执行步数，可以很容易地确定渐进复杂性，步骤是首先确定程序中每条语句（或语句组）的渐进复杂性，然后把这些复杂性加起来。图 2-17至图2-22给出了几个函数的渐进复杂性，但没有提供精确的执行步数分析。

语 句	s/e	频 率	总 步 数
T Sum(T a[], int n)	0	0	$\Theta(0)$
{	0	0	$\Theta(0)$
T tsum = 0;	1	1	$\Theta(1)$
for(int i=0;i<n;i++)	1	n+1	$\Theta(n)$
tsum += a[i];	1	n	$\Theta(n)$
return tsum;	1	1	$\Theta(1)$
}	0	0	$\Theta(0)$

$$t_{\text{sum}}(n)=\Theta(\max\{g_i(n)\})=\Theta(n)$$

图2-17 函数Sum（程序1-8）的渐进复杂性

语 句	s/e	频 率	总 步 数
T Rsum(T a[], int n)	0	0	$\Theta(0)$
{	0	0	$\Theta(0)$
if(n)	1	n+1	$\Theta(n)$
return Rsum(a,n-1) + a[n-1];	1	n	$\Theta(n)$
return 0;	1	1	$\Theta(1)$
}	0	0	$\Theta(0)$

$$t_{\text{Rsum}}(n)=\Theta(n)$$

图2-18 函数Rsum（程序1-9）的渐进复杂性

语 句	s/e	频 率	总 步 数
void Add( T **a, ...)	0	0	$\Theta(0)$
{	0	0	$\Theta(0)$
for (int i = 0; i < rows; i++)	1	$\Theta(\text{rows})$	$\Theta(\text{rows})$
for (int j = 0; j < cols; j++)	1	$\Theta(\text{rows*cols})$	$\Theta(\text{rows*cols})$
c[i][j] = a[i][j] + b[i][j];	1	$\Theta(\text{rows*cols})$	$\Theta(\text{rows*cols})$
}	0	0	$\Theta(0)$

$$t_{\text{Add}}(\text{rows}, \text{cols})=\Theta(\text{rows*cols})$$

图2-19 函数Add（程序2-19）的渐进复杂性

语 句	s/e	频 率	总 步 数
void Transpose(T **a, int rows)	0	0	$\Theta(0)$
{	0	0	$\Theta(0)$
for (int i = 0; i < rows; i++)	1	$\Theta(\text{rows})$	$\Theta(\text{rows})$
for (int j = i+1; j < rows; j++)	1	$\Theta(\text{rows}^2)$	$\Theta(\text{rows}^2)$
Swap(a[i][j], a[j][i]);	1	$\Theta(\text{rows}^2)$	$\Theta(\text{rows}^2)$
}	0	0	$\Theta(0)$

$$t_{\text{Transpose}}(\text{rows}) = \Theta(\text{rows}^2)$$

图2-20 函数Transpose（程序2-22）的渐进复杂性

语 句	s/e	频 率	总 步 数
void Inef(T a[], T b[], int n)	0	0	$\Theta(0)$
{	0	0	$\Theta(0)$
for (int j = 0; j < n; j++)	1	$\Theta(n)$	$\Theta(n)$
b[j] = Sum(a, j + 1);	2j+6	n	$\Theta(n^2)$
}	0	0	$\Theta(0)$

$$t_{\text{Inef}}(n) = \Theta(n^2)$$

图2-21 函数Inef（程序2-23）的渐进复杂性

语 句	s/e	频 率	总 步 数
int SequentialSearch(T a[], T& x, int n)	0	0	$\Theta(0)$
{	0	0	$\Theta(0)$
int i;	1	1	$\Theta(1)$
for (i = 0; i < n && a[i] != x; i++)	1	(0), O(0)	(0), $\Theta(0)$
if (i == n) return -1;	1	1	$\Theta(1)$
return i;	1	(0), O(0)	
}	0	0	$\Theta(0)$

$$t_{\text{SequentialSearch}}(n) = (1)$$

$$t_{\text{SequentialSearch}}(n) = (n)$$

图2-22 函数SequentialSearch（程序2-1）的渐进复杂性

有时可以把 $O(g(n))$ 、 $(g(n))$ 和 $\Theta(g(n))$ 分别解释成如下集合：

$$O(g(n)) = \{f(n) \mid f(n) = O(g(n))\}$$

$$(g(n)) = \{f(n) \mid f(n) = (g(n))\}$$

$$\Theta(g(n)) = \{f(n) \mid f(n) = \Theta(g(n))\}$$

在这种解释下，诸如 $O(g_1(n)) = O(g_2(n))$ 和 $\Theta(g_1(n)) = \Theta(g_2(n))$ 这样的语句就有了明确的含义。在使用这种解释时，可以把 $f(n) = O(g(n))$ 读作“ $f(n)$ 是 $g(n)$ 的一个大O成员”，另两种的读法类似。

按照执行步数来分析图 2-17 至 2-22 时, 可把  $t_p(n) = \Theta(g(n))$ ,  $t_p(n) = O(g(n))$  或  $t_p(n) = \Omega(g(n))$  看成一条程序  $P$  的语句 (用于计算时间), 因为每一步仅需要  $\Theta(1)$  的执行时间。

当有了使用表的经历以后, 就会有要从全局角度来考察程序的渐进复杂性的需要。下面用几个例子来详细地阐述这种方法。

例 2-40 [排列] 考察程序 1-10 中产生排列方式的代码。假定  $m=n-1$ 。当  $k=m$  时, 所需要的时间为  $\Theta(n)$ 。当  $k < m$  时, 将执行 else 语句, 此时, for 循环将被执行  $m-k+1$  次。由于每次循环所花费的时间为  $\Theta(t_{perm}(k+1, m))$ , 因此, 当  $k < m$  时,  $t_{perm}(k, m) = \Theta((m-k+1) t_{perm}(k+1, m))$ 。使用置换的方法, 可以得到:  $t_{perm}(0, m) = \Theta((m+1)*(m+1)!) = \Theta(n*n!)$ , 其中  $n-1$ 。

例 2-41 [折半搜索] 程序 2-30 是一个用来在有序数组  $a[0:n-1]$  中查找元素  $x$  的函数。变量 left 和 right 用来记录查找的起始点和结束点。开始时, 将在 0 到  $n-1$  之间进行查找, 所以 left 和 right 的初值分别为 0 和  $n-1$ 。我们始终遵循以下规律: 当且仅当  $x$  是  $a[\text{left}:\text{right}]$  中的元素时,  $x$  是  $a[0:n-1]$  中的元素。

程序 2-30 折半搜索

```
template<class T>
int BinarySearch(T a[], const T& x, int n)
// 在 a[0] <= a[1] <= ... <= a[n-1] 中搜索 x
// 如果找到, 则返回所在位置, 否则返回 -1
int left = 0; int right = n - 1;
while (left <= right) {
    int middle = (left + right)/2;
    if (x == a[middle]) return middle;
    if (x > a[middle]) left = middle + 1;
    else right = middle - 1;
}
return -1; // 未找到 x
}
```

搜索过程从  $x$  与数组中间元素的比较开始。如果  $x$  等于中间元素, 则查找过程结束。如果  $x$  小于中间元素, 则仅需要查找数组的左半部分, 所以 right 被修改为 middle-1。如果  $x$  大于中间元素, 则仅需要在数组的右半部分进行查找, left 将被修改为 middle+1。这种搜索方法被称为折半搜索 (binary search)。

While 的每次循环 (最后一次除外) 都将以减半的比例缩小搜索的范围, 所以该循环在最坏情况下需执行  $\Theta(\log n)$  次。由于每次循环需耗时  $\Theta(1)$ , 因此, 在最坏情况下, 总的时间复杂性为  $\Theta(\log n)$ 。

例 2-42 [插入排序] 程序 2-15 使用插入排序算法对  $n$  个元素进行排序。对于每个  $i$  值, 最内部的 for 循环在最坏情况下时间复杂性为  $\Theta(1)$ , 因此, 在最坏情况下, 程序 2-15 的时间复杂性最多为  $\Theta(1+2+3+\dots+n) = \Theta(n^2)$ 。在最好的情况下, 程序 2-15 的时间复杂性为  $\Theta(n)$ 。程序的渐进复杂性可由  $\Theta(n)$  和  $O(n^2)$  给出。

小写  $o$  符号通常用于执行步数的分析。执行步数  $3n+O(n)$  表示  $3n$  加上上限为  $n$  的项。在进行这种分析时, 可以忽略步数少于  $\Theta(n)$  的程序部分。

可以扩充 $O$ 、 $\Theta$ 和 $o$ 的定义,采用具有多个变量的函数。例如, $f(n, m) = O(g(n, m))$ 当且仅当存在正常量 $c, n_0$ 和 $m_0$ ,使得对于所有的 $n \geq n_0$ 和所有的 $m \geq m_0$ ,有 $f(n, m) \leq cg(n, m)$ 。

## 练习

28. 仅使用 $O$ 、 $\Theta$ 和 $o$ 的定义来证明如下公式的正确性。不得使用定理 2-1至2-6或图2-15与2-16中的等式。

- 1)  $5n^2 - 6n = \Theta(n^2)$
- 2)  $n! = O(n^2)$
- 3)  $2n^2 2^n + n \log n = \Theta(n^2 2^n)$
- 4)  $\sum_{i=0}^n i^2 = \Theta(n^3)$
- 5)  $\sum_{i=0}^n i^3 = \Theta(n^4)$
- 6)  $n^{2^n} + 6 \cdot 2^n = \Theta(n^{2^n})$
- 7)  $n^3 + 10^6 n^2 = \Theta(n^3)$
- 8)  $6n^3 / (\log n + 1) = O(n^3)$
- 9)  $n^{1.001} + n \log n = \Theta(n^{1.001})$
- 10)  $n^{k+\varepsilon} + n^k \log n = \Theta(n^{k+\varepsilon}), k \geq 0, \varepsilon > 0$

29. 采用定理2-2, 2-4和2-6完成练习28。

30. 证明以下等式不成立:

- 1)  $10n^2 + 9 = O(n)$
- 2)  $n^2 \log n = \Theta(n^2)$
- 3)  $n^2 / \log n = \Theta(n^2)$
- 4)  $n^3 2^n + 6n^2 3^n = O(n^3 2^n)$

31. 证明定理2-3和2-5。

32. 证明定理2-4和2-6。

33. 证明当且仅当 $\lim_{n \rightarrow \infty} f(n)/g(n) = 0$ 时 $f(n) = o(g(n))$ 。

34. 证明图2-15中的等价性E5至E8是正确的。

35. 证明图2-16中的推理规则I5至I6是正确的。

36. 下面哪些规则是正确的?为什么?

- 1)  $\{f(n) = O(F(n)), g(n) = O(G(n))\} \rightarrow f(n)/g(n) = O(F(n)/G(n))$
- 2)  $\{f(n) = O(F(n)), g(n) = O(G(n))\} \rightarrow f(n)/g(n) = \Omega(F(n)/G(n))$
- 3)  $\{f(n) = O(F(n)), g(n) = O(G(n))\} \rightarrow f(n)/g(n) = \Theta(F(n)/G(n))$
- 4)  $\{f(n) = \Omega(F(n)), g(n) = \Omega(G(n))\} \rightarrow f(n)/g(n) = \Omega(F(n)/G(n))$
- 5)  $\{f(n) = \Omega(F(n)), g(n) = \Omega(G(n))\} \rightarrow f(n)/g(n) = O(F(n)/G(n))$
- 6)  $\{f(n) = \Omega(F(n)), g(n) = \Omega(G(n))\} \rightarrow f(n)/g(n) = \Theta(F(n)/G(n))$
- 7)  $\{f(n) = \Theta(F(n)), g(n) = \Theta(G(n))\} \rightarrow f(n)/g(n) = \Theta(F(n)/G(n))$

8)  $\{f(n) = \Theta(F(n)), g(n) = \Theta(G(n))\} \rightarrow f(n)/g(n) = \Omega(F(n)/G(n))$

9)  $\{f(n) = \Theta(F(n)), g(n) = \Theta(G(n))\} \rightarrow f(n)/g(n) = O(F(n)/G(n))$

37. 计算以下函数的渐进复杂性，设计一个类似于图 2-19 至 2-22 的频率表。

1) Factorial(见程序 1-7)

2) MinMax(见程序 2-26)

3) MinMax(见程序 2-27)

4) Mult(见程序 2-24)

5) Mult(见程序 2-25)

6) Max(见程序 1-31)

7) PolyEval(见程序 2-3)

8) Horner(见程序 2-4)

9) Rank(见程序 2-5)

10) Perm(见程序 1-10)

11) SelectionSort(见程序 2-7)

12) SelectionSort(见程序 2-12)

13) InsertionSort(见程序 2-14)

14) InsertionSort(见程序 2-15)

15) BubbleSort(见程序 2-9)

16) BubbleSort(见程序 2-13)

## 2.5 实际复杂性

我们已经知道，一个程序的时间复杂性通常是其实例特征的函数，在确定程序的时间需求是如何随着实例特征的变化而变化时，这种函数将非常有用。我们也可以利用复杂性函数对两个执行相同任务的程序  $P$  和  $Q$  进行比较。假定程序  $P$  具有复杂性  $\Theta(n)$ ，程序  $Q$  具有复杂性  $\Theta(n^2)$ ，由此可以断定，对于“足够大”的  $n$ ，程序  $P$  比程序  $Q$  快。为了说明这种推断的正确性，可以通过实际考察。对于某些常量  $c$  和所有的  $n$ ， $n \geq n_1$ ，程序  $P$  的实际计算时间的上限为  $cn$ ；对于某些常量  $d$  和所有的  $n$ ， $n \geq n^2$ ，程序  $Q$  的实际计算时间的下限为  $dn^2$ 。由于对于所有  $n \geq c/d$ ，有  $cn \leq dn^2$ ，因此每当  $n \geq \max\{n_1, n_2, c/d\}$  时，程序  $P$  比程序  $Q$  快。

我们应该谨慎地使用上面推断中“足够大”的说法。在决定使用两个程序中的哪个程序时，必须了解所要处理的  $n$  是否真的足够大。如果程序  $P$  的实际运行时间为  $10^6 n$  毫秒，程序  $Q$  的实际运行时间为  $n^2$  毫秒，若总有  $n \geq 10^6$ ，那么优先使用的将是程序  $Q$ 。

为了能体会各种函数是如何随着  $n$  的增长而变化的，可以仔细地研究图 2-23 和 2-24。从图中可以看出，随着  $n$  的增长， $2^n$  的增长极快。事实上，如果程序需要  $2^n$  执行步，那么当  $n=40$  时，执行步数将大约为  $1.1 \times 10^{12}$ 。在一台每秒执行 1 000 000 000 步的计算机中，该程序大约需要执行 18.3 分钟；如果  $n=50$ ，同样的程序在该台机器上将需要执行 13 天，当  $n=60$  时，需要执行 310.56 年；当  $n=100$  时，则需要执行  $4 \times 10^{13}$  年。因此可以认定，具有指数复杂性的程序仅适合于小的  $n$  (典型地取  $n \leq 40$ )。

具有高次多项式复杂性的函数也必须限制使用。例如，如果程序需要  $n^{10}$  执行步，那么当  $n=10$  时，每秒执行 1 000 000 000 步的计算机需要 10 秒钟；当  $n=100$  时，需要 3171 年； $n=1000$  时，将需要  $3.17 \times 10^{13}$  年。如果程序的复杂性是  $n^3$ ，则当  $n=1000$  时，需要执行 1 秒； $n=10\ 000$  时，需



要110.67分钟； $n=100\ 000$ 时，需要11.57天。

$\log n$	$n$	$n \log n$	$n^2$	$n^3$	$2^n$
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4 096	65 536
5	32	160	1 024	32 768	4 294 967 296

图2-23 各种函数的值

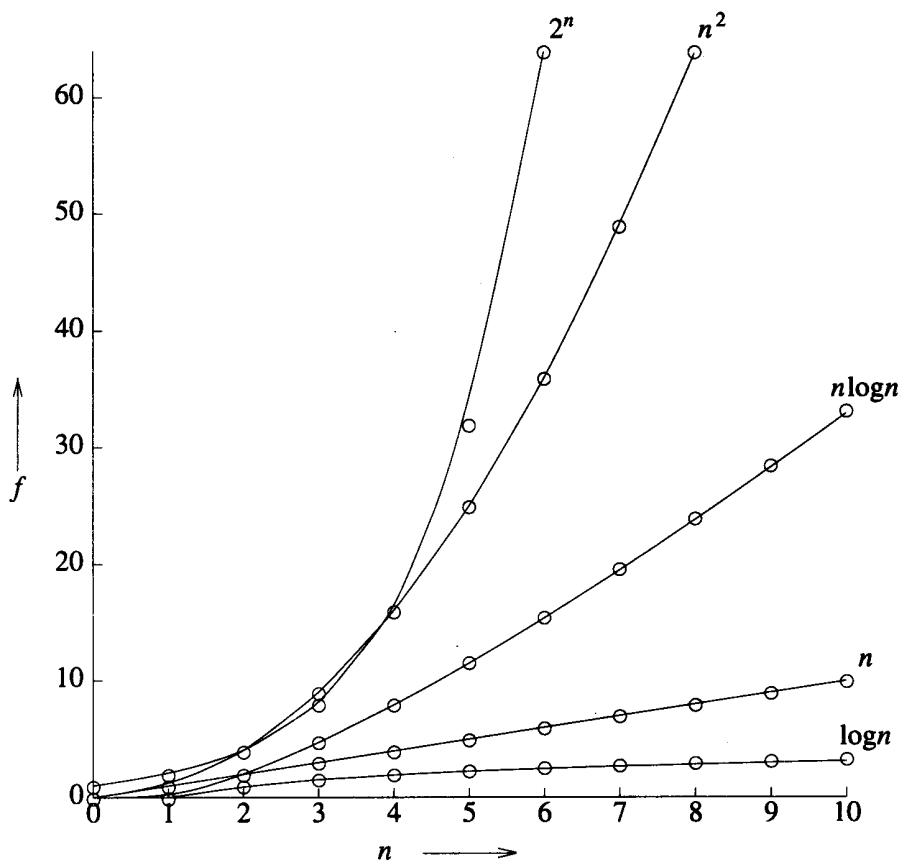


图2-24 各种函数的曲线

图2-25中给出了复杂性为 $f(n)$ 的程序在每秒执行1 000 000 000条指令的计算机上执行时所需要的时间。应该注意到，目前只有世界上最快的计算机才能每秒执行1 000 000 000条指令。从实际应用来看，对于相当大的 $n$ （比如 $n > 100$ ），仅那些复杂性比较小（如 $n$ ， $n \log n$ ， $n^2$ ， $n^3$ ）的程序才是可行的，即使能够制造出每秒执行 $10^{12}$ 条指令的计算机。如果有这样的计算机，图2-25中的计算时间将分别减小1000倍。如 $n=100$ 时，执行 $n^{10}$ 条指令需耗时3.17年，执行 $2^n$ 条指令需耗时 $4 \times 10^{10}$ 年。

n	f(n)						
	n	$n \log_2 n$	$n^2$	$n^3$	$n^4$	$n^{10}$	$2^n$
10	.01us	.03us	.1us	1us	10us	10s	1us
20	.02us	.09us	.4us	8us	160us	2.84h	1ms
30	.03us	.15us	.9us	27us	810us	6.83d	1s
40	.04us	.21us	1.6us	64us	2.56ms	121d	18m
50	.05us	.28us	2.5us	125us	6.25ms	3.1y	13d
100	.10us	.66us	10us	1ms	100ms	3171y	$4 \times 10^{13}y$
$10^3$	1us	9.96us	1ms	1s	16.67m	$3.17 \times 10^{13}y$	$32 \times 10^{283}y$
$10^4$	10us	130us	100ms	16.67m	115.7d	$3.17 \times 10^{23}y$	
$10^5$	100us	1.66ms	10s	11.57d	3171y	$3.17 \times 10^{33}y$	
$10^6$	1ms	19.92ms	16.67m	31.17y	$3.17 \times 10^7y$	$3.17 \times 10^{43}y$	

us=微秒= $10^{-6}$ 秒；ms=毫秒= $10^{-3}$ 秒；s=秒；m=分钟；h=小时；d=天；y=年

图2-25 每秒1 000 000 000条指令的机器上所需要的执行时间

## 练习

38. 设A和B是执行相同任务的程序，令 $t_A(n)$ 和 $t_B(n)$ 分别表示它们的运行时间。对于下面的每对数据，给出使程序A比程序B快的 $n$ 的取值范围。

1)  $t_A(n)=1000n$ ,  $t_B(n)=10n^2$

2)  $t_A(n)=2n^2$ ,  $t_B(n)=n^3$

3)  $t_A(n)=2^n$ ,  $t_B(n)=100n$

4)  $t_A(n)=1000n \log_2 n$ ,  $t_B(n)=n^2$

39. 假定一台计算机每秒能执行1万亿条指令，重新给出图2-25中的数据。

40. 假定有某个程序和某台计算机，它们可以在“合理的时间”内解决规模为 $n=N$ 的问题。建立一个表格来说明，对于同样的程序和速度快 $x$ 倍的计算机，能在合理的时间内解决的问题的最大规模是多少（即最大的 $n$ 值）。分别取 $x=10$ 、100、1000、1 000 000及 $t_A(n)=n$ 、 $n^2$ 、 $n^3$ 、 $n^5$ 和 $2^n$ 来完成练习。

## 2.6 性能测量

性能测量（performance measurement）主要关注于得到一个程序实际需要的空间和时间。按照前几节的说明，这些量与特定的编译器及编译器选项密切相关，同时也与执行程序的计算机密切相关。除非特别声明，本书中的所有性能值都是在486/DX50 PC机上，用Borland C++5.01 for Windows95及缺省的编译器选项获得的。

我们忽略编译所需要的时间和空间是因为每个程序仅需编译一次（当然是在调试完成之后），而可以运行无数次。不过，如果测试的次数要比运行最终代码的次数多，则在程序测试期间，编译所需要的时间和空间也是很重要的。

基于以下原因，我们不能精确地测量一个程序运行时所需要的时间和空间：

- 指令空间和数据空间的大小是由编译器在编译时确定的，所以不必测量这些数据。
- 采用前几节介绍的方法，可以很准确地估算递归栈空间和变量动态分配所需要的空间。

为了得到一个程序的执行时间，需要一个定时机制。大多数C++产品都提供了相应的计时

函数。例如，Borland C++在它的time.h头文件中定义了clock()函数，该函数用于返回自程序启动以来所流逝的“滴答”数。把流逝的“滴答”数除以常量CLK\_TCK，可以得到流逝的秒数。在PC计算机上，CLK\_TCK=18.2。

假如希望测量程序2-15中的函数InsertionSort 在最坏情况下所需要的时间，首先需要：

- 1) 确定需要测定其执行时间的n值
- 2) 对于上面的每个n，给出能导致最坏复杂性的测试数据。

### 2.6.1 选择实例的大小

可以根据以下两个因素来确定使用哪些n值：程序执行的时间及执行的次数。假定希望预测在最坏情况下，使用插入排序的方法对n个元素的数组进行排序所需要的时间。从例2-42中了解到，在最坏情况下，InsertionSort函数的复杂性为 $\Theta(n^2)$ ，即n的平方函数。在理论上，如果知道任何三种n值所对应的执行时间，就可以确定这个平方函数，该函数可用来描述InsertionSort函数在最坏情况下的执行时间。利用所得到的平方函数，可以得到任何其他n值所对应的执行时间。在实践过程中，通常需要3个以上的n值，其原因如下：

- 1) 渐进分析仅给出了对于足够大的n值时程序的复杂性。对于小的n值，程序的运行时间可能并不满足渐进曲线。为了确定渐进曲线以外的点，需要使用多个n值。
- 2) 即使在满足渐进曲线的区间内，程序实际运行时间也可能不满足预定的渐进曲线，原因是在进行渐进分析时，忽略了许多低层次的时间需求。例如，一个程序的渐进复杂性为 $\Theta(n^2)$ ，而它的实际复杂性可以是 $c_1 n^2 + c_2 n \log n + c_3 n + c_4$ ，或其他任何最高项为 $c_1 n^2$ 的函数，其中 $c_1$ 为常量且 $c_1 > 0$ 。

对于程序2-15，我们只期望获得 $n < 100$ 的渐进复杂性。所以，对于 $n > 100$ 的情况，可能只需要很少量的估算值。一个合理的选择是 $n = 200, 300, 400, \dots, 1000$ ，这个选择并没有任何奥妙，也可以使用 $n = 500, 1000, 1500, \dots, 10000$ 或 $n = 512, 1024, 2048, \dots, 2^{15}$ ，后者将耗费更多的计算时间，而且会得到一些无法想象的时间值（相当巨大）。

对于 $[0, 100]$ 范围内的n值，可以进行更精细的测量，因为我们并不很清楚渐进复杂性从何处开始有效。当然，如果测量结果表明，平方函数复杂性并不是从这个范围开始有效的，那么可对 $[100, 200]$ 范围进行更细致的测量，如此进行下去，直到找到起始点。测算 $[0, 100]$ 范围内的运行时间可以从 $n=0$ 开始，此后，n每次递增10。

### 2.6.2 设计测试数据

对于许多程序，可以手工或用计算机来产生能导致最好和最坏复杂性的测试数据。然而，对于平均的复杂性，通常很难设计相应的数据。如对于InsertionSort，对任何n来说，能导致最坏复杂性的测试数据应是一个递减的序列，如 $n, n-1, n-2, \dots, 1$ ；导致最好复杂性的测试数据应是一个递增的序列，如 $0, 1, 2, \dots, n-1$ 。我们很难提供一组测试数据能使InsertionSort函数表现出平均的复杂性。

当不能给出能产生预期的复杂性的测试数据时，可以根据一些随机产生的测试数据所测量出的最小（最大，平均）时间来估计程序的最坏（最好，平均）复杂性。

### 2.6.3 进行实验

在确定了实例的大小并给出了测试数据以后，就可以编写程序来测量所期望的运行时间了。

对于插入排序，程序2-31给出了测试的过程。相应的测试数据见图2-26。

程序2-31 导致插入排序出现最坏复杂性的程序

```
#include <iostream.h>
#include <time.h>
#include "insort.h"
void main(void)
{
    int a[1000], step = 10;
    clock_t start, finish;
    for (int n = 0; n <= 1000; n += step) {
        // 获得对应于 n 值的时间
        for (int i = 0; i < n; i++)
            a[i] = n - i; // 初始化
        start = clock( );
        InsertionSort(a, n);
        finish = clock( );
        cout << n << ' ' << (finish - start) / CLK_TCK << endl;
        if (n == 100) step = 100;
    }
}
```

图2-26给出了这样的结论：排序 100 个元素以内的数组不需要任何时间，排序 500~600 个元素的数组所花费的时间是一样的。这个结论当然是错误的。对于其他的  $n$  值也有这种异常。问题出在对于计时函数 `clock()` 来说，所需要的运行时间太小。而且，所有测量的精确度均为一个时钟“滴答”。由于在我们的计算机中， $\text{CLK\_TCK} = 18.2$ ，因此测量的误差范围将是一个“滴答”时间  $1/18.2=0.055$  秒。对于  $n=1000$ ，测试程序所报告的时间为 6 个“滴答”，因而实际的执行时间介于 5 到 7 个“滴答”之间。如果希望测量误差在 10% 以内， $\text{finish}-\text{start}$  至少应为 10 个时钟“滴答”或 0.55 秒，这时所得到的结果将与图 2-26 不同。

n	时间 ( s )	n	时间 ( s )
0	0	100	0
10	0	200	0.054945
20	0	300	0
30	0	400	0.054945
40	0	500	0.10989
50	0	600	0.109890
60	0	700	0.164835
70	0	800	0.164835
80	0	900	0.274725
90	0	1000	0.32967

图2-26 程序2-31的测试结果

为了提高测量的精确度，对于每个  $n$  值，可以重复排序若干次。由于排序会改变数组  $a$ ，因此需要在每次排序之前对该数组进行初始化。程序 2-32 给出了新的测试程序。注意，现在所测量的时间为排序的时间、对  $a$  进行初始化的时间以及  $\text{while}$  循环所需要的额外时间。图 2-27 给出了实际的测量时间。

$n$	重复次数	总时间 (s)	每次排序时间 (s)
0	34228	0.549451	0.000016
10	10365	0.549451	0.000053
20	3525	0.549451	0.000156
30	1701	0.549451	0.000323
40	992	0.549451	0.000554
50	647	0.549451	0.000849
60	454	0.549451	0.001210
70	337	0.549451	0.001630
80	259	0.549451	0.002121
90	206	0.549451	0.002667
100	167	0.549451	0.003290
200	43	0.549451	0.012778
300	19	0.549451	0.028918
400	11	0.549451	0.049950
500	7	0.549451	0.078493
600	5	0.604396	0.120879
700	4	0.604396	0.151099
800	3	0.659341	0.219780
900	3	0.769231	0.256410
1000	2	0.604396	0.302198

图2-27 程序2-32的输出结果

通过注解程序 2-32 中的语句 `InsertionSort(a, n)`，然后运行程序 2-32，可以获得  $\text{while}$  循环及初始化数组  $a$  所需要的时间。图 2-28 中给出了对于所选择的  $n$  值，所得到的实际运行时间。从图 2-27 的每个时间中减去图 2-28 中的额外时间，可以得到 `InsertionSort` 在最坏情况下的运行时间。可以看出，由于  $\text{while}$  的条件实际被测试  $\text{counter}+1$  次，而  $\text{while}$  循环体只执行了  $\text{counter}$  次，因此，不精确性依然存在。不过，由于对于较小的  $n$  重复的次数很多，因此这种额外的开销可以被忽略。注意，对于较大的  $n$ ，随着  $n$  的加倍，图 2-27 中的时间相应地将变成 4 倍。这种情形是我们所期望的，因为程序最坏的复杂性为  $\Theta(n^2)$ 。

程序2-32 误差在 10% 以内的测试程序

```
#include <iostream.h>
#include <time.h>
#include "insort.h"
void main(void)
{
```

```

int a[1000], n, i, step = 10;
long counter;
float seconds;
clock_t start, finish;
for (n = 0; n <= 1000; n += step) {
    // 获得对应于 n 值的时间
    start = clock(); counter = 0;
    while (clock() - start < 10) {
        counter++;
        for (i = 0; i < n; i++)
            a[i] = n - i; // 初始化
        InsertionSort(a, n);
    }
    finish = clock();
    seconds = (finish - start) / CLK_TCK;
    cout << n << ' ' << counter << ' ' << seconds << ' ' << seconds / counter << endl;
    if (n == 100) step = 100;
}

```

n	重 复 次 数	总时间 ( s )	每次排序时间 ( s )
0	36141	0.549451	0.000015
10	32321	0.549451	0.000017
50	19186	0.549451	0.000029
100	12999	0.549451	0.000042
500	3557	0.549451	0.000154
1000	1864	0.549451	0.000295

图2-28 图2-27测量中的额外开销

## 练习

41. 为什么程序2-31 的误差范围不在10%以内。

42. 利用程序2-32来获取两个不同的插入排序程序（见程序2-14和程序2-15）在最坏情况下所需要的运行时间。采用与程序2-32相同的n 值。试比较调用Insert函数以及把Insert函数的代码合并到排序函数这两种情况下各自的优缺点。

43. 利用程序2-32来获取两个不同的冒泡排序程序（见程序2-9和程序2-13）在最坏情况下所需要的运行时间。采用与程序2-32相同的n 值。不过，你必须证实，程序2-32中所给出的测试数据实际上也能使这两种冒泡排序函数产生最坏的复杂性。使用三列表格来给出你的结果，三列分别是：n、程序2-9、程序2-13。指出在最坏情形下，这两种冒泡排序函数的相对性能。

44. 1) 对于程序2-7和程序2-12中所给出的两个选择排序函数，设计能产生最坏复杂性的测试数据。

2) 对2-32进行适当的修改以测定这两种选择排序函数在最坏情况下所需要的时间。采用与

程序2-32相同的 $n$ 值。

3) 使用三列表格来给出你的结果，三列分别是： $n$ 、程序2-7、程序2-12。

4) 指出在最坏情形下，这两种选择排序函数的相对性能。

45. 比较插入排序（见程序2-34）、选择排序（见程序2-12）和冒泡排序（见程序2-13）在最坏情形下所需要的运行时间。为了一致，把程序2-13重写为一个函数。

1) 设计能使每种函数产生最坏复杂性的测试数据。

2) 使用1中的数据及程序2-32中的测试程序来获取最坏情况下的运行时间。

3) 采用两种形式来描述这些时间：一是采用一个四列的表格，四列分别是： $n$ 、选择排序、冒泡排序、插入排序；二是采用一个显示三条曲线的图（每条曲线对应一种排序方法），图的 $x$ 轴代表 $n$ 值， $y$ 轴代表时间值。

4) 通过三种排序函数在最坏情形下的性能比较，能得出什么结论？

5) 对于每个 $n$ ，测量额外的时间，并用图2-28的表格形式给出测试结果。从2所得到的时间中减去这种额外开销，然后给出一个新的时间表和新的图。

6) 在减去额外开销后，在4中所得到的结论是否发生了变化？

7) 利用已得到的数据，估计每种排序函数对2000、4000和10000个元素进行排序，在最坏情况下所需要的时间。

46. 修改程序2-32，以便获得InsertionSort函数（见程序2-15）的平均运行时间。要求如下：

1) 在每一次while循环中，对0, 1, ...,  $n-1$ 的随机排列进行排序，这种随机排列是由一个随机排列产生器产生的。如果找不到这样的函数，可以用随机数生成器来编写，或简单地产生一个 $n$ 个数的随机序列。

2) 设置while循环，使得在一次循环中至少有20个随机排列被排序，并且至少需要耗费10个时钟“滴答”。

3) 用耗费的时间除以所排序的随机排列数目，得到平均排序时间。

47. 利用练习46中的策略来估算程序2-9和2-13中给出的冒泡排序函数的平均运行时间。采用与程序2-32相同的 $n$ 值。用表格形式给出测试结果。

48. 利用练习46中的策略来估算程序2-7和2-12中给出的选择排序函数的平均运行时间，采用与程序2-32相同的 $n$ 值。用表格形式给出测试结果。

49. 利用练习46中的策略来估算并比较程序2-12、程序2-13和2-15中所给出的排序函数的平均运行时间，采用与程序2-32相同的 $n$ 值。分别用表格和图的形式给出测试结果。

50. 编写测试程序来确定顺序搜索（见程序2-1）和折半搜索（见程序2-30）在搜索成功时所需要的平均时间。假定数组中每个元素被搜索的概率相同。用表格和图的形式给出结果。

51. 编写测试程序来确定顺序搜索（见程序2-1）和折半搜索（见程序2-30）在搜索成功时，最坏情况下所需要的时间。用表格和图的形式给出结果。

52. 对于 $n=10, 20, 30, \dots, 100$ ，确定函数Add（见程序2-19）的运行时间。用表格和图的形式给出测量结果。

53. 对于 $n=10, 20, 30, \dots, 100$ ，确定函数Transpose（见程序2-22）的运行时间。用表格和图的形式给出测量结果。

54. 对于 $n=10, 20, 30, \dots, 100$ ，确定函数Mult（见程序2-24）的运行时间。用表格和图的形式给出测量结果。



## 2.7 参考及推荐读物

下面的参考书给出了若干程序的渐进分析：

- 1) E.Horowitz, S.Sahni, S.Rajasekaran. *Fundamentals of Computer Algorithms/C++*. W.H.Freeman, 1997。
- 2) E.Horowitz, S.Sahni, D.Mehta. *Fundamentals of Data Structures in C++*. W.H.Freeman, 1995。
- 3) T.Cormen, C.Leiserson, R.Rivest. *Introduction to Algorithms*. McGraw-Hill, 1992。
- 4) G.Rawlins. *Compared to What: An Introduction to the Analysis of Algorithms*. W.H.Freeman, 1992。
- 5) B.Moret, H.Shapiro. *Algorithms from P to NP* 第1卷: *Design and Efficiency*. Benjamin-Cummings, 1991。

## 第二部分 数据结构

### 第3章 数据描述

从本章开始我们进行数据结构的研究，一直到第12章为止。尽管本章的重点是介绍线性表，但一个主要目标是为了使大家明白，数据可以用不同的形式进行描述或存储在计算机存储器中。最常见的数据描述方法有：公式化描述、链接描述、间接寻址和模拟指针。

公式化描述借助数学公式来确定元素表中的每个元素分别存储在何处（如存储器地址）。最简单的情形就是把所有元素依次连续存储在一片连续的存储空间中，这就是通常所说的连续线性表。

在链接描述中，元素表中的每个元素可以存储在存储器的不同区域中，每个元素都包含一个指向下一个元素的指针。同样，在间接寻址方式中，元素表中的每个元素也可以存储在存储器的不同区域中，不同的是，此时必须保存一张表，该表的第  $i$  项指向元素表中的第  $i$  个元素，所以这张表是一个用来存储元素地址的表。

在公式化描述中，元素地址是由数学公式来确定的；在链接描述中，元素地址分布在每一个表元素中；而在间接寻址方式下，元素地址则被收集在一张表中。

模拟指针非常类似于链接描述，区别在于它用整数代替了 C++ 指针，整数所扮演的角色与指针所扮演的角色完全相同。

本章介绍了线性表的四种描述形式，通过考察常见表操作（如插入、删除）的复杂性，给出了每种描述方法的优缺点。在本章中还将学习如何用数组来模拟 C++ 指针。

本章所给出的有关数据结构的概念如下：

- 抽象数据类型。
- 公式化描述、链接描述、间接寻址和模拟指针。
- 单向链表、循环链表和双向链表。

本章的应用部分集中介绍了链表的应用，之所以这样做，是因为第1章和第2章中所给出的所有程序都采用了公式化的数据表示形式，而现在希望采用链表形式来改写其中的部分程序。二叉排序、基数排序和等价类应用都使用了链表，而凸面体应用则采用了双向链表。二叉排序和基数排序可用来对  $n$  个元素进行排序，如果关键值介于一个“合适的范围”内，排序所需要的时间为  $\Theta(n)$ 。虽然在第2章中所给出的排序算法需要耗时  $O(n^2)$ ，但它不需要将关键值限制在一个“合适的范围”内。当关键值介于一个“合适的范围”内时，二叉排序和基数排序要比第2章所给出的排序算法快出许多。在二叉排序的应用中还可以看到如何把函数名作为一个参数传递给 C++ 函数。

#### 3.1 引言

数据对象（data object）即一组实例或值，例如：

- $Boolean = \{false, true\}$

- $Digit = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $Letter = \{A, B, C, \dots, Z, a, b, \dots, z\}$
- $NaturalNumber = \{0, 1, 2, \dots\}$
- $Integer = \{0, \pm 1, \pm 2, \pm 3, \dots\}$
- $String = \{a, b, \dots, aa, ab, ac, \dots\}$

$Boolean$ ,  $Digit$ ,  $Letter$ ,  $NaturalNumber$ ,  $Integer$ 和 $String$ 都是数据对象,  $true$ 和 $false$ 是 $Boolean$ 的实例, 而 $0, 1, \dots$ , 和 $9$ 都是 $Digit$ 的实例。数据对象的每个实例要么是一个原语 (primitive) (或原子 (atomic)), 要么是由其他数据对象的实例复合而成。在后一种情形下, 用术语“元素 (element)”来表示对象实例的单个组件。

例如, 数据对象 $NaturalNumber$ 的每个实例均可以视为原子, 在这种情况下, 不必考虑对这些实例做进一步的分解。另一种观点是把 $NaturalNumber$ 的每个实例看成是由 $Digit$ 数据对象的若干实例复合而成。按照这种观点, 整数 $675$ 是由数字 $6, 7$ 和 $5$ 按顺序组成。

数据对象 $String$ 是所有可能的串实例的集合, 每个实例均由字符组成。串实例的一些具体例子如: *good*, *a trip to Hawaii*, *going down hill*和*abcbcdabcde*。其中第一个串由四个元素  $g, o, o$  和  $d$  按序构成, 而每个元素又都是数据对象 $Letter$ 的一个实例。数据对象的实例以及构成实例的元素通常都有某种相关性, 例如, 自然数 $0$ 是最小的自然数,  $1$ 是仅比 $0$ 大的自然数, 而 $2$ 是 $1$ 之后的下一个自然数。在自然数 $675$ 中,  $6$ 是最高有效位,  $7$ 在其次, 而 $5$ 是最低有效位。在串 $good$ 中,  $g$ 是第一字母,  $o$ 是第二和第三个字母, 而 $d$ 是最后一个字母。

除了相互关联之外, 对于任一个数据对象通常都有一组相关的函数。这些函数可以把对象的某个实例转换成该对象的另外一个实例, 或转换成另一个数据对象的实例, 或者同时进行上述两种转换。函数也可以简单地创建一个新的实例, 而不必对一个新创建的实例进行转换。例如, 进行自然数相加的函数将创建一个新的自然数, 该自然数是两个相加数之和, 两个参与加操作的自然数本身不会发生任何变化。

数据结构 (data structure) 包括数据对象和实例以及构成实例的每个元素之间所存在的各种关系。这些关系可由相关的函数来实现。

当我们研究数据结构时, 关心的是数据对象 (实际上是实例) 的描述以及与数据对象相关函数的具体实现。数据对象的良好描述可以有效地促进函数的高效实现。

最常使用的数据对象以及函数都已经在 C++ 中被当作标准的数据类型加以实现, 如整数对象 (int)、实数对象 (float)、布尔对象 (bool) 等。所有其他的数据对象均可以采用标准数据类型、枚举类型以及由 C++ 的类、数组和指针特性所提供的组合功能来描述。例如, 可以用一个字符数组  $s$  来描述  $String$  的实例:

```
char s[MaxSize];
```

有关数据结构的研究包括两个部分。本章按照数据的各种描述方法进行组织, 即基于公式的描述、链表描述、简接寻址和模拟指针。我们利用数据对象线性表来演示这些方法。在后续章节中将研究其他流行的数据描述方法, 如矩阵、栈、队列、字典、优先队列和图。

## 3.2 线性表

线性表 (linear list) 是这样的数据对象, 其实例形式为:  $(e_1, e_2, \dots, e_n)$ , 其中  $n$  是有穷自然数。  $e_i$  是表中的元素,  $n$  是表的长度。元素可以被视为原子, 因为它们本身的结构与线性表的结构无关。当  $n = 0$  时, 表为空; 当  $n > 0$  时,  $e_1$  是第一个元素,  $e_n$  是最后一个元素, 可以认为  $e_1$  优先于  $e_2$ ,  $e_2$  优先于  $e_3$ , 如此等等。除了这种优先关系之外, 在线性表中不再有其他结构。

我们用  $s$  表示每个元素  $e_i$  所需要的字节数，因此， $s$  是一个元素的大小。

以下是一些线性表的例子：1) 一个班级学生姓名按字母顺序排列的列表；2) 按递增次序排列的考试分数表；3) 按字母顺序排列的会议列表；4) 奥林匹克男子篮球比赛中金牌获得者按年代次序排列的列表。根据这些例子可知对于线性表有必要执行下列操作：

- 创建一个线性表。
- 确定线性表是否为空。
- 确定线性表的长度。
- 查找第  $k$  个元素。
- 查找指定的元素。
- 删除第  $k$  个元素。
- 在第  $k$  个元素之后插入一个新元素。

可以用一个抽象数据类型 (abstract data type, ADT) 来说明线性表，它给出了实例及相关操作的描述 (见 ADT 3-1)。

请注意，这种抽象数据类型说明与 C++ 的类定义之间具有很多相似性。抽象数据类型说明独立于我们已提到的任何描述方法。抽象数据类型的描述方法必须满足抽象数据类型说明，而说明又反过来保证了描述方法的合法性。除此以外，所有满足说明的描述方法都可以在数据类型应用中替换使用。

#### ADT 3-1 线性表的抽象数据类型描述

抽象数据类型 *LinearList* {

实例

0或多个元素的有序集合

操作

*Create* (): 创建一个空线性表

*Destroy* (): 删除表

*IsEmpty*(): 如果表为空则返回 true，否则返回 false

*Length* (): 返回表的大小 (即表中元素个数)

*Find* ( $k, x$ ): 寻找表中第  $k$  个元素，并把它保存到  $x$  中；如果不存在，则返回 false

*Search* ( $x$ ): 返回元素  $x$  在表中的位置；如果  $x$  不在表中，则返回 0

*Delete* ( $k, x$ ): 删除表中第  $k$  个元素，并把它保存到  $x$  中；函数返回修改后的线性表

*Insert* ( $k, x$ ): 在第  $k$  个元素之后插入  $x$ ；函数返回修改后的线性表

*Output* ( $out$ ): 把线性表放入输出流  $out$  之中

}

除了用 ADT 3-1 中非正式的自然语言来说明抽象数据类型外，还可以使用 C++ 的抽象类来说明抽象数据类型。在该方法中需使用派生类、抽象类和虚拟函数，我们将在第 5 章和第 12 章中详细讨论这些话题，目前还只能使用非正式的自然语言。如果你已经很熟悉派生类和抽象类，可以查阅 12.9.4 节看看如何用 C++ 抽象类来说明线性表抽象数据类型。

## 3.3 公式化描述

### 3.3.1 基本概念

公式化描述 (formula-based) 采用数组来表示一个对象的实例，数组中的每个位置被称之为

为单元 ( cell ) 或节点 ( node ), 每个数组单元应该足够大, 以便能够容纳数据对象实例中的任意一个元素。在某些情况下, 每个实例可分别用一个独立的数组来描述, 而在其他情况下, 可能要使用一个数组来描述几个实例。实例中每个元素在数组中的位置可以用一个数学公式来指明。

假定使用一个数组来描述表, 需要把表中的每个元素映射到数组的具体位置上。第一个元素在什么地方? 第二个元素在什么地方? 在公式化描述中, 可用一个数学公式来确定每个元素的位置。一个简单的映射公式如下:

$$location(i) = i - 1 \quad (3-1)$$

公式 ( 3-1 ) 指明表中第  $i$  个元素 ( 如果存在的话 ) 位于数组中  $i-1$  位置处。图 3-1a 给出了一个利用公式 ( 3-1 ) 作为映射公式, 在数组 element 中描述一个 5 元素线性表的例子。( 一个更简洁的公式是:  $location(i)=i$ , 它不使用 0 位置, 在练习 8 至练习 13 中将使用这个公式 )。

为了完整地描述线性表, 需要了解表的当前长度或大小, 为此, 使用变量 length 作为表的长度。当表为空时, length 为 0。程序 3-1 给出了相应的 C++ 类定义。由于表元素的数据类型随着应用的变化而变化, 所以定义了一个模板类, 在该模板类中, 用户指定元素的数据类型为 T。数据成员 length、MaxSize 和 element 都是私有成员, 其他成员均为共享成员。Insert 和 Delete 均返回一个线性表的引用。我们将要看到, 具体实现时首先会修改表 \*this, 然后返回一个引用 ( 指向修改后的表 )。因此, 同时组合多个表操作是可行的, 如 X.Insert(0,a).Delete(3,b)。

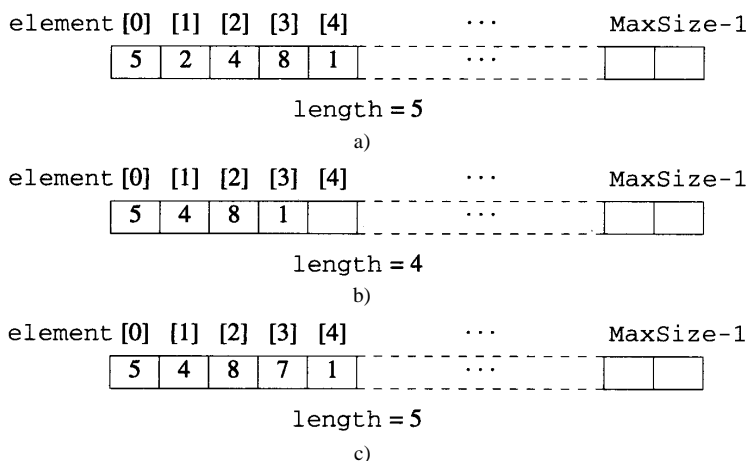


图3-1 线性表

程序3-1 基于公式的类 LinearList

```
template<class T>
class LinearList {
public:
    LinearList(int MaxListSize = 10); //构造函数
    ~LinearList() {delete [] element;} //析构函数
    bool IsEmpty() const {return length == 0;}
    int Length() const {return length;}
    bool Find(int k, T& x) const; //返回第k个元素至x中
    int Search(const T& x) const; // 返回x所在位置
```

```
LinearList<T>& Delete(int k, T& x); // 删除第k个元素并将它返回至x中
LinearList<T>& Insert(int k, const T& x); // 在第k个元素之后插入x
void Output(ostream& out) const;
private:
    int length;
    int MaxSize;
    T *element; // 一维动态数组
};
```

### 3.3.2 异常类NoMem

如果分配内存失败，本书中所给出的大多数代码都能引发一个异常。有时，异常可能由new引起，而有时则需要我们自己来引发。我们希望在所有情形下都能引发同一个异常，因此，定义了一个异常类NoMem，见程序3-2。函数my\_new\_handler简单地引发一个类型为NoMem的异常。程序3-2的最后一行调用了C++函数set\_new\_handler，每当分配内存失败时，该函数就让操作符new调用函数my\_new\_handler。所以，new引发的是异常NoMem而不是xalloc。每当分配内存失败时，set\_new\_handler将返回一个指针，指向由new此前所调用的那个函数，该指针保存在变量Old\_Handler\_中。为了恢复new的原始行为，可以作如下调用：

```
set_new_handler(Old_Handler_);
```

注意，程序3-2中用一个函数名作为实参，另外，不管new以前引发的是哪一种类型的异常，程序3-2都将把new引发的异常变成NoMem异常。

程序3-2 使new引发NoMem异常而不是xalloc异常

```
// 内存不足
class NoMem {
public:
    NoMem () {}
};
// 使new引发NoMem异常而不是xalloc异常
void my_new_handler()
{
    throw NoMem();
}
new_handler Old_Handler_=set_new_handler(my_new_handler);
```

### 3.3.3 操作

操作Create和Destroy分别被作为类的构造函数和析构函数加以实现。构造函数（见程序3-3）创建一个最大缺省长度为10的表。当计算机没有足够的内存来创建一个所期望长度的数组时，操作符new引发一个类型为NoMem的异常，构造函数代码并没有捕获这个异常。如果所引发的异常没有在程序的任何位置被捕获，程序将非正常终止。在大多数代码中，我们都希望应用代码能够捕获所引发的异常。

程序3-3 基本的表操作

```
template<class T>
```

```

LinearList<T>::LinearList(int MaxListSize)
{
    // 基于公式的线性表的构造函数
    MaxSize = MaxListSize;
    element = new T[MaxSize];
    length = 0;
}

template<class T>
bool LinearList<T>::Find(int k, T& x) const
{
    // 把第k个元素取至x中
    // 如果不存在第k个元素则返回false, 否则返回true
    if (k < 1 || k > length) return false; // 不存在第k个元素
    x = element[k - 1];
    return true;
}

template<class T>
int LinearList<T>::Search(const T& x) const
{
    // 查找x, 如果找到, 则返回x所在的位置
    // 如果x不在表中, 则返回0
    for (int i = 0; i < length; i++)
        if (element[i] == x) return ++i;
    return 0;
}

```

下面的语句创建一个整数线性表y, 其最大长度为100:

```
LinearList<int> y(100);
```

析构函数(见程序3-1)调用delete以便释放由构造函数分配给数组element的空间。程序3-1中包含了IsEmpty和Length的代码, 而程序3-3给出了Find和Search的代码。IsEmpty、Length和Find的复杂性为 $\Theta(1)$ , 而Search的复杂性为 $O(\text{length})$ 。

为了从一个表中删除第k个元素, 需要首先确认表中包含第k个元素, 然后再删除这个元素。如果表中不存在第k个元素, 则出现一个异常。ADT LinearList(见ADT 3-1)没有告诉我们这个时候该做什么。我们的代码将引发一个类型为 OutOfBounds的异常。每当正在执行的函数中任一参数超出所期望的范围时, 就引发这种类型的异常。

如果存在第k个元素, 可以将元素  $k+1, k+2, \dots, \text{length}$  依次向前移动一个位置, 并将length的值减1, 从而删除第k个元素。例如, 为了从图3-1a的表中删除第二个元素, 需要把元素4, 8和1分别移动到表的1, 2和3位置处, 这些位置分别对应数组element的1、2和3位置。图3-1b给出了删除第二个元素之后的表, 这时, 表的长度为4。

当线性表按照公式(3-1)进行描述时, 函数Delete(见程序3-4)给出了删除操作。如果不存在第k个元素, 将引发一个异常, Delete所需要的时间为 $\Theta(1)$ ; 如果存在第k个元素, 则移动length-k个元素, 需要耗时 $\Theta((\text{length}-k)s)$ , 其中s是每个元素的大小。此外, 被删除的元素被移动至x, 因此, 总的时间复杂性为 $O((\text{length}-k)s)$ 。

程序3-4 从线性表中删除一个元素

```

template<class T>
LinearList<T>& LinearList<T>::Delete(int k, T& x)
{
    // 把第k个元素放入x中, 然后删除第k个元素

```



```
// 如果不存在第k个元素，则引发异常 OutOfBounds
if (Find(k, x)) { // 把元素 k+1, ...向前移动一个位置
    for (int i = k; i < length; i++)
        element[i-l] = element[i];
    length--;
    return *this;
}
else throw OutOfBounds();
}
```

为了在表中第k个元素之后插入一个新元素，首先需要把 k+1至length元素向后移动一个位置，然后把新元素插入到 k+1位置处。例如，在图 3-1b 的表中第三个元素之后插入 7，将得到图3-1c 的结果。程序 3-5中给出了完整的、插入一个新元素的 C++代码。可以注意到，在插入操作期间，可能出现两类异常。第一类异常发生的情形是：没有正确指定插入点，如在插入新元素之前，表中元素个数少于 k-1个，或者 k<0。在这种情形下，引发一个 OutOfBounds异常。当表已经满时，会发生第二类异常，此时，数组没有剩余的空间来容纳新元素，因此将引发一个NoMem异常。Insert的时间复杂性为  $O((length-k)s)$ 。

程序3-5 向线性表中插入一个元素

```
template<class T>
LinearList<T>& LinearList<T>::Insert(int k, const T& x)
{ // 在第k个元素之后插入x
    // 如果不存在第k个元素，则引发异常 OutOfBounds
    // 如果表已经满，则引发异常 NoMem
    if (k < 0 || k > length) throw OutOfBounds();
    if (length == MaxSize) throw NoMem();
    //向后移动一个位置
    for (int i = length-i; i >= k; i--)
        element[i+l] = element[i];
    element[k] = x;
    length++;
    return *this;
}
```

程序3-6给出了Output的代码，其时间复杂性为  $\Theta(length)$ 。这段代码简单地把表元素插入到输出流out之中。为了实际显示线性表，可以重载操作符<<，见程序3-6。

程序3-6 把线性表输送至输出流

```
template<class T>
void LinearList<T>::Output(ostream& out) const
{ //把表输送至输出流
    for (int i = 0; i < length; i++)
        out << element[i] << " ";
}
// 重载 <<
template <class T>
```

```
ostream& operator<<(ostream& out, const LinearList<T>& x)
{x.Output(out); return out;}
```

程序3-7是一个使用类LinearList的C++程序，它假定程序3-1至3-6均存储在文件l1ist.h之中，且异常类定义位于文件xcept.h之中。该示例完成如下工作：创建一个大小为5的整数线性表L；输出该表的长度（为0）；在第0个元素之后插入2；在第一个元素之后插入6（至此，线性表为2，6）；寻找并输出第一个元素（为2）；输出当前表的长度（为2）；删除并输出第一个元素。图3-2给出了由程序3-7产生的输出。

程序3-7 采用类LinearList的例子

```
#include <iostream.h>
#include "l1ist.h"
#include "xcept.h"
void main(void)
{
    try {
        LinearList<int> L(5);
        cout << "Length = " << L.Length() << endl;
        cout << "IsEmpty = " << L.IsEmpty() << endl;
        L.Insert(0,2).Insert(1,6);
        cout << "List is " << L << endl;
        cout << "IsEmpty = " << L.IsEmpty() << endl;
        int z;
        L.Find(1,z);
        cout << "First element is " << z << endl;
        cout << "Length = " << L.Length() << endl;
        L.Delete(1,z);
        cout << "Deleted element is " << z << endl;
        cout << "List is " << L << endl;
    }
    catch (...) {
        cerr << "An exception has occurred" << endl;
    }
}
```

```
Length = 0
IsEmpty = 1
List is 2 6
IsEmpty = 0
First element is 2
Length = 2
Deleted element is 2
List is 6
```

图3-2 程序3-7所产生的输出

### 3.3.4 评价

在接受一个线性表的公式化描述方法之前，先来考察一下这种描述方法的优缺点。的确，对于一个线性表的各种操作可以用非常简单的 C++ 函数来实现。执行查找、删除和修改的函数都有一个最差的、与表的大小呈线性关系的时间复杂性。我们可能很满足于这种复杂性。（在第7章和第11章将看到能够更快地执行这些操作的描述方法。）

这种描述方法的一个缺点是空间的低效利用。考察如下情形：我们需要维持三个表，而且已经知道在任何时候这三个表所拥有的元素总数都不会超过 5000 个。然而，很有可能在某个时刻一个表就需要 5000 个元素，而在另一时刻另一个表也需要 5000 个元素。若采用类 `LinearList`，这三个表中的每一个表都需要有 5000 个元素的容量。因此，即使我们在任何时刻都不会使用 5000 以上的元素，也必须为此保留总共 15 000 个元素的空间。

为了避免这种情形，必须把所有的线性表都放在一个数组 `list` 中进行描述，并使用两个附加的数组 `first` 和 `last` 对这个数组进行索引。图 3-3 给出了在一个数组 `list` 中描述的三个线性表。我们采用大家很习惯的约定，即如果有  $m$  个表，则每个表从 1 到  $m$  进行编号，且 `first[i]` 为第  $i$  个表中的第一个元素。有关 `first[i]` 的约定使我们更容易地采用公式化描述方式。`last[i]` 是表  $i$  的最后一个元素。注意，根据这些约定，每当第  $i$  个表不为空时，有 `last[i] > first[i]`，而当第  $i$  个表为空时，有 `last[i] = first[i]`。所以在图 3-3 的例子中，表 2 是空表。在数组中，各线性表从左至右按表的编号次序 1, 2, 3, ...,  $m$  进行排列。

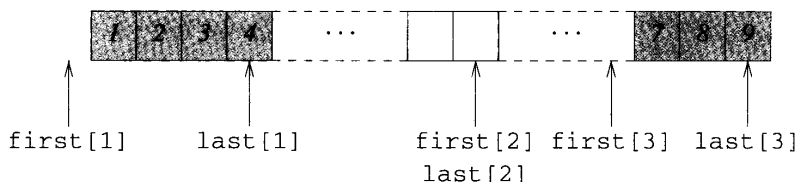


图3-3 一个数组中的所有线性表

为了避免第一个表和最后一个表的处理方法与其他的表不同，定义了两个边界表：表 0 和表  $m+1$ ，其中 `first[0] = last[0] = -1`，`first[m+1] = last[m+1] = MaxSize - 1`。

为了在第  $i$  个表的第  $k$  个元素之后插入一个元素，首先需要为新元素创建空间。如果 `last[i] = first[i+1]`，则在第  $i$  个表和第  $i+1$  个表之间没有空间，因此不能把第  $k+1$  至最后一个元素向后移动一个位置。在这种情况下，通过检查关系式 `last[i-1] < first[i]` 是否成立，可以确定是否有可能把第  $i$  个表的 1 至  $k-1$  元素向前移一个位置；如果这个关系式不成立，要么需要把表 1 至表  $i-1$  的元素向前移一个位置，要么把表  $i+1$  至表  $m$  向后移一个位置，然后为表  $i$  创建需要增长的空间。当表中所有的元素总数少于 `MaxSize` 时，这种移位的方法是可行的。

图 3-4 是一个伪 C++ 函数，它向表  $i$  中插入一个新的元素，可以把该函数细化为兼容的 C++ 代码。

尽管在一个数组中描述几个线性表比每个表用一个数组来描述空间的利用率更高，但在最坏的情况下，插入操作将耗费更多的时间。事实上，一次插入操作可能需要移动 `MaxSize - 1` 个元素。

### 练习

1. 类 `LinearList` 的一个缺点是需要预测线性表最大可能的尺寸，解决的方法之一是：在创

```
int insert(int i, int k, int y)
{ //在表i 的第k 个元素之后插入 y
  // first和last是全局数组
  int j, m;
  m = last[i] - first[i]; //表i中的元素数
  if (k < 0 || k > m) return 0;
  //在右边有剩余空间吗？
  寻找最小的j (j > i), 使得 last [j] < first [j+1];
  如果存在这样的j, 则把表i+1至表j以及表i的k+1至最末元素均向后移动一个位置,
  然后将y插入i中;
  这种移动需要相应地修改last和first的值
  // 在左边有剩余空间吗？
  如果不存在上述的j, 则寻找另一个最大的j (j<i), 使得last [j] < first [j+1];
  如果找到这样的j, 则把表j至表i-1以及表i的1至k-1元素均向前移动一个位置,
  然后将y插入i中;
  这种移动需要相应地修改last和first的值
  // 成功否？
  return ( (没有找到上述的j)?0: 1);
}
```

图3-4 向一个数组中（包含多个线性表）插入一个元素的伪代码

建线性表时，置MaxSize=1，之后在执行插入操作期间，如果在表中已经有了MaxSize个元素，则将MaxSize加倍，按照这个新尺寸分配一个新数组，并将老数组中的数据复制到新数组中，最后将老数组删除。类似地，在执行删除操作期间，如果线性表的尺寸降至当前MaxSize的四分之一，则分配一个更小的、尺寸为MaxSize/2的数组，并将老数组中的数据复制到新数组中，最后将老数组删除。

1) 采用上述思想，重新实现类LinearList。构造函数不应带参数，并将MaxSize置为1，并分配一个大小为1的数组，同时置length=0。

2) 考虑从一个空表开始，对其连续实施 $n$ 次表操作。假定在使用以前的实现方法时用了 $f(n)$ 步，试证明对于上述新的实现方法，存在常数 $c$ ，使得执行步数最多为 $cf(n)$ 。

2. 假设一个线性表的描述满足公式（3-1）

1) 扩充LinearList类的定义，增加一个函数Reverse，该函数将表中元素的次序变反。反序操作是就地进行的（即在数组element本身的空间内）。注意，在反序操作进行之前，表中第 $k$ 个元素（如果存在）位于element[k-1]，完成反序之后，该元素位于element[length-k]。

2) 证明上述函数的复杂性与线性表的长度成线性关系。

3) 采用适当的测试数据来验证代码的正确性。

4) 请编写另外一个就地处理的反序函数，它能对LinearList类型的对象进行反序操作。该函数不是LinearList类的成员函数，但它应利用成员函数来产生反序线性表。

5) 上述函数的时间复杂性是多少？

6) 分别采用大小为1000，5000和10 000的线性表来比较以上两种反序函数的执行效率。

3. 扩充LinearList类的定义，增加一个成员函数Half()。调用X.Half()将删除X中半数的元素。比如，如果X.length的初始值为7，且X.element[]=[2, 13, 4, 5, 17, 8, 29]，则执行X.Half()后，

X.length的值为4，且X.element[]=[2, 4, 17, 29]；如果X.length的初始值为4，且X.element[]=[2, 13, 4, 5]，则执行X.Half()后，X.length的值为2，且X.element[]=[2, 4]。如果X开始时是空表，则执行X.Half()后，X仍然为空表。

1) 试给出成员函数Half()的代码。不能利用任何其他的LinearList成员函数。代码的复杂性应为 $\Theta(\text{length})$ 。

2) 证明代码的复杂性确实为 $\Theta(\text{length})$ 。

3) 使用适当的测试数据来测试代码的正确性。

4. LinearList缺省的复制构造函数仅复制length, MaxSize和element的值。因此，当用线性表L作为函数F的实际参数（对应于值参X）来调用F时，L.length, L.MaxSize和L.element被复制到X的相应成员中。当函数F退出时，LinearList的析构函数被X唤醒，数组X.element（与数组L.element相同）被删除。避免L.element被删除的一种方法是定义复制构造函数如下：

```
LinearList<T>::LinearList(const LinearList<T>& L)
```

该函数复制length和MaxSize的值，然后创建一个新数组element，并将L.element[0:MaxSize-1]复制到element中。试编写这个复制构造函数，并估计其复杂性。

5. 在许多应用中，需要对一个线性表中的元素进行前移和后移操作。试扩充LinearList类的定义，增加一个私有成员变量current，它纪录线性表当前的位置。此外，需要增加的共享成员如下：

1) Reset—置current为1。

2) Current(x)—返回x中的当前元素。

3) End—当且仅当当前元素为表的最后一个元素时，返回true。

4) Front—当且仅当当前元素为表的第一个元素时，返回true。

5) Next—移动current至表中的下一个元素，如果操作失败则引发一个异常。

6) Previous—移动current至表中的前一个元素，如果操作失败则引发一个异常。

试编写上述代码，并使用适当的测试数据来测试代码的正确性。

6. 设A和B均为LinearList对象

1) 编写一个新的成员函数Alternate(A,B)以创建一个新的线性表，该表包含了A和B中的所有元素，其中A和B的元素轮流出现，表中的首元素为A中的第一个元素。在轮流排列元素时，如果某个表的元素用完了，则把另一个表的其余元素依次添加在新表的后部。代码的复杂性应与两个输入表的长度呈线性比例关系。

2) 证明代码具有线性复杂性。

3) 使用适当的测试数据来测试代码的正确性。

7. 设A和B均为LinearList对象。假定A和B中的元素都是按序排列的（如从左至右按递增次序排列）。

1) 试编写一个成员函数Merge(A, B)，用以创建一个新的有序线性表，该表中包含了A和B的所有元素。

2) 考察所编写的函数的时间复杂性。

3) 用适当的测试数据来测试代码的正确性。

8. 1) 试编写函数LinearList::Split(A, B)，该函数用来创建两个线性表A和B，A中包含\*this的所有奇数元素（注意一个线性表的奇数元素是指element偶数位置上的元素），B中包含其余的元素。

2) 考察函数的时间复杂性。

3) 用适当的测试数据来测试代码的正确性。

9. 假定采用如下公式来描述一个线性表：

$$location(i) = i \quad (3-2)$$

1) 对于程序3-1中的类定义需要做相应的修改吗？如果是，请编写新的定义。

2) 能够描述的最长的表的长度是多少？

3) 修改程序3-1中的所有函数，以满足公式 (3-2)

4) 用适当的测试数据来测试代码的正确性。

5) 每个函数的时间复杂性分别是多少？

10. 用公式 (3-2) 代替公式 (3-1) 来完成练习2。

11. 用公式 (3-2) 代替公式 (3-1) 来完成练习6。

12. 用公式 (3-2) 代替公式 (3-1) 来完成练习7。

13. 用公式 (3-2) 代替公式 (3-1) 来完成练习8。

14. 假定采用下述公式来描述一个线性表：

$$location(i) = (location(1) + i - 1) \% MaxSize \quad (3-3)$$

其中MaxSize是用来存储表元素的数组的大小。与专门保留一个表长的做法不同的是，用变量first 和last 来指出表的第一个元素和最后一个元素的位置。

1) 基于该公式，设计一个与LinearList相似的类。

2) first和last的初始值应该是多少？

3) 对于新的类，试编写出所有成员函数的代码。（通过适当的选择，将元素移动到欲插入/删除元素的左边或右边，可以编写出更高效的Delete和Insert代码。）

4) 考察每个函数的时间复杂性。

5) 用适当的测试数据来测试代码的正确性。

15. 用公式 (3-3) 代替公式 (3-1) 来完成练习2。

16. 用公式 (3-3) 代替公式 (3-1) 来完成练习6。

17. 用公式 (3-3) 代替公式 (3-1) 来完成练习7。

18. 用公式 (3-3) 代替公式 (3-1) 来完成练习8。

19. 将图3-4细化为C++ 函数，并测试其正确性。

20. 编写一个C++ 函数，该函数在表 $i$ 的第 $k$ 个元素之后插入一个元素。假定在一个数组中存放了 $n$ 个线性表。如果不得不移动多个表以便容纳新元素，新的函数应该首先确定可用空间的数量。移动表时应保证每个表所包含的可用空间数量应大体相同，以便适应未来增长的需要。通过编译和执行，测试代码的正确性。

21. 编写一个C++函数，该函数用来从表 $i$ 中删除第 $k$ 个元素。假定在一个数组中存放了 $n$ 个线性表。通过编译和执行，测试代码的正确性。

## 3.4 链表描述

### 3.4.1 类ChainNode 和Chain

在链表描述中，数据对象实例的每个元素都放在单元或节点中进行描述。不过，节点不必是一个数组元素，因此没有什么公式可用来定位某个元素。取而代之的是，每个节点中都包含了与该节点相关的其他节点的位置信息。这种关于其他节点的位置信息被称之为链（link）或

指针 ( pointer )。

令  $L=(e_1, e_2, \dots, e_n)$  是一个线性表。在针对该表的一个可能的链表描述中，每个元素  $e_i$  都放在不同的节点中加以描述。每个节点都包含一个链接域，用以指向表中的下一个元素。所以节点  $e_i$  的指针将指向  $e_{i+1}$ ，其中  $1 \leq i < n$ 。节点  $e_n$  没有下一个节点，所以它的链接域为 NULL(或0)。指针变量  $first$  指向描述中的第一个节点。图3-5给出了表  $L=(e_1, e_2, \dots, e_n)$  的链表描述。

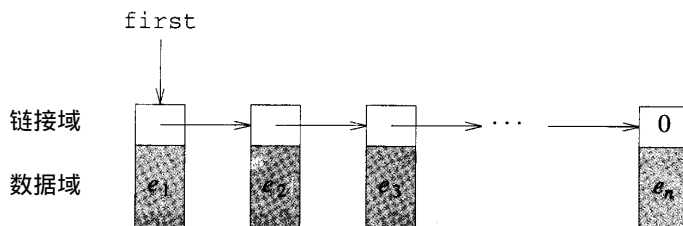


图3-5 线性表的链表描述

由于图3-5中的每个链表节点都正好有一个链接域，所以该图的链表结构被称之为单向链表 ( singly linked list )。并且，由于第一个节点  $e_1$  的指针指向第二个节点  $e_2$ ， $e_2$  的指针指向  $e_3$ ，...，最后一个节点链接域为 NULL(或0)，故这种结构也被称作链 ( chain )。为了把一个线性表表示成一个链，可以使用程序 3-8 中的类定义 ChainNode 和 Chain。由于 Chain<T> 是 ChainNode<T> 的一个友类，所以 Chain<T> 可以访问 ChainNode<T> 的所有成员 ( 尤其是私有成员 )。共享成员 Length、Find、Delete 和 Insert 的定义与程序 3-1 完全一致。

程序3-8 链表的类定义

```
template <class T>
class ChainNode {
    friend Chain<T>;
private:
    T data;
    ChainNode<T> *link;
};

template<class T>
class Chain {
public:
    Chain() {first = 0;}
    ~Chain();
    bool IsEmpty() const {return first == 0;}
    int Length() const;
    bool Find(int k, T& x) const;
    int Search(const T& x) const;
    Chain<T>& Delete(int k, T& x);
    Chain<T>& Insert(int k, const T& x);
    void Output(ostream& out) const;
private:
    ChainNode<T> *first; // 指向第一个节点的指针
};
```



### 3.4.2 操作

可以采用如下的描述来创建一个空的整数型线性表：

```
Chain<int> L;
```

注意，线性表的链表描述不需要指定表的最大长度。

程序3-9给出了析构函数的代码，它的复杂性为  $\Theta(n)$ ，其中  $n$  为链表的长度。程序3-10和程序3-11中的代码分别实现了 Length 操作和 Find 操作。Length 的复杂性为  $\Theta(n)$ ，Find 的复杂性为  $O(k)$ 。函数 Search(见程序3-12)假定对于类型  $T$  定义了  $!=$  操作，该函数的复杂性为  $O(n)$ 。Output(见程序3-13)的复杂性为  $\Theta(n)$ ，它要求对于类型  $T$  必须定义  $<<$  操作。

程序3-9 删除链表中的所有节点

---

```
template<class T>
Chain<T>::~~Chain()
{// 链表的析构函数，用于删除链表中的所有节点
    ChainNode<T> *next; // 下一个节点
    while (first) {
        next = first->link;
        delete first;
        first = next;
    }
}
```

---

程序3-10 确定链表的长度

---

```
template<class T>
int Chain<T>::Length() const
{// 返回链表中的元素总数
    ChainNode<T> *current = first;
    int len = 0;
    while (current) {
        len++;
        current = current->link;
    }
    return len;
}
```

---

程序3-11 在链表中查找第  $k$  个元素

---

```
template<class T>
bool Chain<T>::Find(int k, T& x) const
{// 寻找链表中的第  $k$  个元素，并将其传送到  $x$ 
// 如果不存在第  $k$  个元素，则返回 false，否则返回 true
    if (k < 1) return false;
    ChainNode<T> *current = first;
    int index = 1; // current 的索引
    while (index < k && current) {
        current = current->link;
```

---

```

    index++;
}
if (current) {x = current->data; return true;}
return false; // 不存在第k个元素
}

```

程序3-12 在链表中搜索

```

template<class T>
int Chain<T>::Search(const T& x) const
// 寻找x，如果发现x，则返回x的地址
//如果x不在链表中，则返回0
    ChainNode<T> *current = first;
    int index = 1; // current的索引
    while (current && current->data != x) {
        current = current->link;
        index++;
    }
    if (current) return index;
    return 0;
}

```

程序3-13 输出链表

```

template<class T>
void Chain<T>::Output(ostream& out) const
// 将链表元素送至输出流
    ChainNode<T> *current;
    for (current = first; current; current = current->link)
        out << current->data << " ";
}
//重载<<
template <class T>
ostream& operator<<(ostream& out, const Chain<T>& x)
    {x.Output(out); return out;}

```

为了从图3-6所示的链中删除第四个元素，需进行如下操作：

- 1) 找到第三和第四个节点。
- 2) 使第三个节点指向第五个节点。

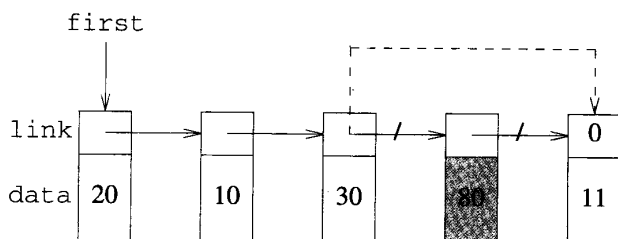


图3-6 删除第四个节点

3) 释放第四个节点所占空间，以便于重用。

程序3-14中给出了删除操作的代码。有三种情形需要考虑。第一种情形是： $k$ 小于1或链表为空；第二种情形是：第一个元素将被删除且链表不为空；最后一种情形是：从一个非空的链表中删除首元素之外的其他元素。

程序3-14 从链表中删除一个元素

```
template<class T>
Chain<T>& Chain<T>::Delete(int k, T& x)
{// 把第k个元素取至x，然后从链表中删除第k个元素
//如果不存在第k个元素，则引发异常 OutOfBounds
    if (k < 1 || !first)
        throw OutOfBounds(); // 不存在第k个元素
    // p最终将指向第k个节点
    ChainNode<T> *p = first;
    // 将p移动至第k个元素，并从链表中删除该元素
    if (k == 1) // p已经指向第k个元素
        first = first->link; // 删除之
    else ( // 用q指向第k-1个元素
        ChainNode<T> *q = first;
        for (int index = 1; index < k - 1 && q; index++)
            q = q->link;
        if (!q || !q->link)
            throw OutOfBounds(); //不存在第k个元素
        p = q->link; // 存在第k个元素
        q->link = p->link;) // 从链表中删除该元素
    //保存第k个元素并释放节点p
    x = p->data;
    delete p;
    return *this;
}
```

程序3-14的代码首先处理第一种情形，即引发 OutOfBounds 异常。对于其他两种情形，定义了一个指针变量  $p$ ，并将它初始化为指向链表中的第一个元素。如果  $k$  是1，则  $p$  将指向链表中的第  $k$  个节点，语句  $\text{first}=\text{first}->\text{link}$  则用来从链表中删除  $p$  所指向的节点。如果  $k$  大于1，指针变量  $q$  用来定位第  $k-1$  个节点，定位过程可采用一个 for 循环，假定链表中有多个节点，则当从 for 循环中退出时， $q$  会指向第  $k-1$  个节点。如果链表的节点数少于  $k-1$ ，则  $q$  为0。接下来的 if 语句首先判断  $q$  是否为0，如果  $q$  不为0，则 if 语句通过检查  $q->\text{link}$  来判断链表中是否存在第  $k$  个元素。如果 if 语句的判断结果为真，则将修改  $p$  指针以使其指向第  $k$  个节点。通过将前一个节点（即  $q$  节点）改为指向  $p$  的下一个节点，即可将第  $k$  个节点从链表中删除。

当从 if-else 结构中退出时， $p$  指向第  $k$  个节点，并且该节点已经脱离链表。下面只需把  $p$  的数据域传递给参数  $x$ ，并释放节点  $p$  所占用的空间。

为了检验程序3-14的正确性，分别用一个空表和一个至少包含一个节点的链表进行测试。此外，还可对不同的  $k$  值进行测试，如  $k=0$ 、 $k=n$ 、 $k=n$ 、 $0 < k < n$  等，其中  $n$  为链表长度。

插入和删除的过程很相似。为了在链表的第  $k$  个元素之后插入一个新元素，需要首先找到第  $k$  个元素，然后在该节点的后面插入新节点。图 3-7 给出了  $k=0$  和  $k=0$  两种情况下链表指针的变化。插入之前的实线指针，在插入之后被“打断”。程序3-15给出了相应的 C++ 代码，它

的复杂性为 $O(k)$ 。尽管插入操作的代码中引用了`new`，但它没有去捕获可能产生的异常，而是由`Insert`的调用者捕获。

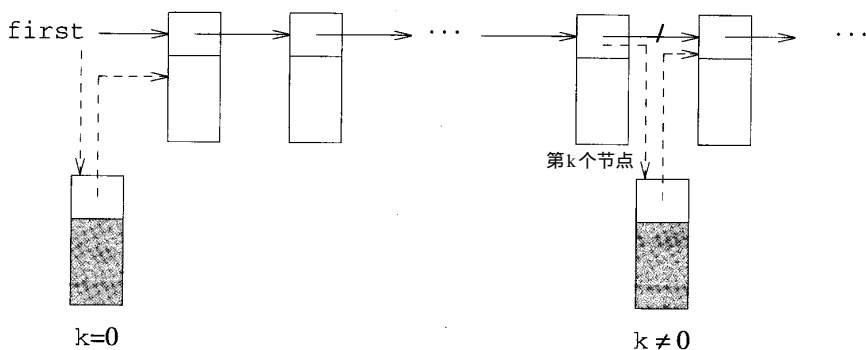


图3-7 向链表中插入元素

程序3-15 向链表中插入元素

```
template<class T>
Chain<T>& Chain<T>::Insert(int k, const T& x)
{// 在第k个元素之后插入x
//如果不存在第k个元素，则引发异常OutOfBounds
// 如果没有足够的空间，则传递 NoMem异常
if (k < 0) throw OutOfBounds();
// p 最终将指向第k个节点
ChainNode<T> *p = first;
//将p移动至第k个元素
for (int index = 1; index < k && p; index++)
    p = p->link;
if (k > 0 && !p) throw OutOfBounds(); //不存在第k个元素
// 插入
ChainNode<T> *y=new ChainNode<T>;
y->data = x;
if (k) {// 在p之后插入
    y->link = p->link;
    p->link = y;}
else {// 作为第一个元素插入
    y->link = first;
    first = y;}
return *this;
}
```

### 3.4.3 扩充类Chain

在某些链表的应用中，希望执行一些未在抽象数据类型 `LinearList`(见ADT3-1)中出现的那些操作。因此有必要扩充`Chain`的类定义，以便包含一些附加的函数，如`Erase`(删除链表中的所有节点)、`Zero`(将`first`指针置为0，但并不删除任何节点)、`Append`(在链表的尾部添加一个元素)。函数`Erase`(见程序3-16)等价于类的析构函数。事实上，根据`Erase`的定义，可以把类的析构函数

简单地定义为对Erase的调用。

程序3-16 删除链表中的所有节点

---

```
template<class T>
void Chain<T>::Erase()
{//删除链表中的所有节点
    ChainNode<T> *next;
    while (first) {
        next = first->link;
        delete first;
        first = next;}
}
```

---

函数Zero可以定义为如下的内联函数：

```
void Zero() { first = 0;}
```

为了在⊙(1)的时间内添加一个元素，需要借助一个新的、类型为 ChainNode<T> \*的私有成员last来跟踪链表的最后一个元素。程序 3-17给出了 Append的代码，为了使该段代码能正确运行，必须在Delete函数（见程序3-14）的语句

```
p = q -> link
```

之后添加如下语句：

```
if ( p == last ) last = q ;
```

在Insert函数（见程序3-15）的语句return \*this之前必须加入下面的语句：

```
if (!y -> link ) last = y;
```

程序3-17 在链表右端添加一个元素

---

```
template < class T >
Chain < T > & Chain < T > ::Append(const T& x)
{//在链表尾部添加x
    ChainNode< T > *y;
    y = new ChainNode< T >;
    y->data = x; y->link = 0;
    if (first) {//链表非空
        last->link = y;
        last = y;}
    else // 链表为空
        first = last = y;
    return *this;
}
```

---

#### 3.4.4 链表遍历器类

暂且假定Output不是Chain类的成员函数，并且在该类中没有重载操作符 <<。为了输出链表X，将不得不执行如下代码：

```
int len = X.Length();
for (int i = 1; i <= len; i++) {
```

```
X.Find(i,x);
cout << X << ' ' ;}
```

这段代码的复杂性为  $\Theta(n^2)$ ，而成员函数 Output 的复杂性为  $\Theta(n)$ ，其中  $n$  为链表长度。类似于 Output 函数，许多使用链的应用代码都要求从链表的第一个元素开始，从左至右依次检查每一个元素。采用遍历器（Iterator）可以大大方便这种从左至右的检查，遍历器的功能是纪录当前位置并每次向前移动一个位置。

链表遍历器（见程序 3-18）有两个共享成员 Initialize 和 Next。Initialize 返回一个指针，该指针指向第一个链表节点中所包含的数据，同时把私有变量 location 设置为指向链表的第一个节点，该变量用来跟踪我们在链表中所处的位置。成员 Next 用来调整 location，使其指向链表中的下一个节点，并返回指向该节点数据域的指针。由于 ChainIterator 类访问了 Chain 类的私有成员 first，所以应把它定义为 Chain 的友类。

程序 3-18 链表遍历器类

```
template<class T>
class ChainIterator {
public:
    T* Initialize(const Chain<T>& c)
    {location = c.first;
    if (location) return &location->data;
    return 0;}
    T* Next()
    {if (!location) return 0;
    location = location->link;
    if (location) return &location->data;
    return 0;}
private:
    ChainNode<T> *location;
};
```

像前面一样，假定 Output 不是 Chain 类的成员函数，并且在该类中没有重载操作符 <<。采用链表遍历器，可以在线性时间内输出链表，见程序 3-19。

程序 3-19 采用链表遍历器输出整数链表 X

```
int *x;
ChainIterator<int> c;
x = c.Initialize(X);
while (x) {
    cout << *x << ' ' ;
    x = c.Next();
}
cout << endl;
```

### 3.4.5 循环链表

采纳下面的一条或两条措施，使用链表的应用代码可以更简洁、更高效：1) 把线性表描述成一个单向循环链表（singly linked circular list），或简称循环链表（circular list），而不是一个

单向链表；2) 在链表的前部增加一个附加的节点，称之为头节点（head node）。通过把单向链表最后一个节点的链接指针改为指向第一个节点，就可以把一个单向链表改造成循环链表，如图3-8a 所示。图3-8b 给出了一个带有头指针的非空的循环链表，图3-8c 给出了一个带有头指针的空循环链表。

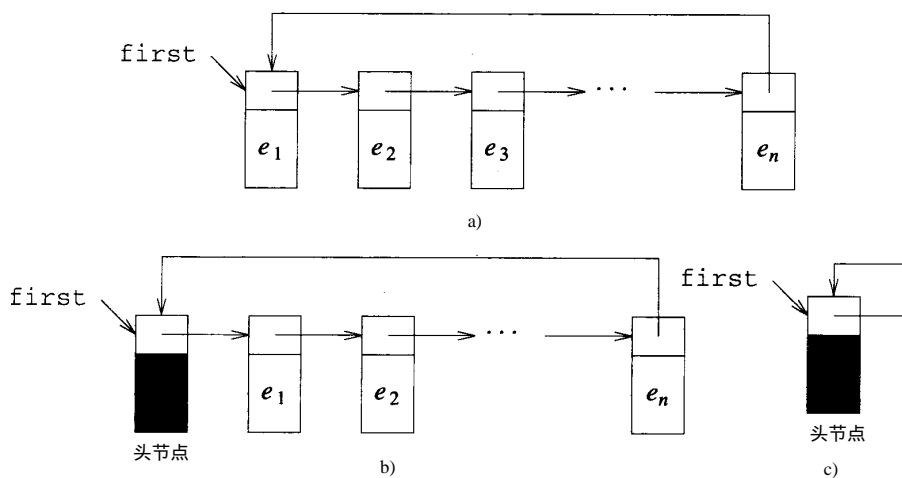


图3-8 循环链表

a) 循环链表 b) 带有头节点的循环链表 c) 空表

在使用链表时，头指针的使用非常普遍，因为利用头指针，通常可以使程序更简洁、运行速度更快。CircularList类的定义与Chain类的定义很类似。尽管链表搜索的复杂性仍然保持为 $O(n)$ ，但代码本身要稍微简单一些。由于与程序3-12相比，程序3-20在for循环的每次循环中执行的比较次数较少，因此程序3-20将比程序3-12运行得更快一些，除非要查找的元素紧靠链表的左部。

程序3-20 在带有头节点的循环链表中进行查找

```
template<class T>
int CircularList<T>::Search(const T& x) const
{
    // 在带有头节点的循环链表中寻找 x
    ChainNode<T> *current = first->link;
    int index = 1; // current的索引
    first->data = x; // 把x放入头节点
    // 查找x
    while (current->data != x) {
        current = current->link;
        index++;
    }
    // 是链表表头吗？
    return ((current == first) ? 0 : index);
}
```

### 3.4.6 与公式化描述方法的比较

采用公式化描述方法的线性表仅需要能够保存所有元素的空间以及保存表长所需要的空



间，而链表和循环链表描述还需要额外的空间，用来保存链接指针（线性表中的每个元素都需要一个相应的链接指针）。采用链表描述所实现的插入和删除操作要比采用公式化描述时执行得更快。当每个元素都很长时（字节数多），尤其如此。

还可以使用链接模式来描述很多表，这样做并不会降低空间利用率，也不会降低执行效率。对于公式化描述，为了提高空间利用率，不得不把所有的表都放在一个数组中加以描述，并使用了另外两个数组来对这个数组进行索引，更有甚者，与一个表对应一个数组的情形相比，插入和删除操作变得更为复杂，而且存在一个很显著的最坏运行时间。

采用公式化描述，可以在  $O(1)$  的时间内访问第  $k$  个元素。而在链表中，这种操作所需要的时间为  $O(k)$ 。

### 3.4.7 双向链表

对于线性表的大多数应用来说，采用链表和 / 或循环链表已经足够了。然而，对于有些应用，如果每个链表元素既有指向下一个元素的指针，又有指向前一个元素的指针，那么在设计应用代码时将更为方便。双向链表（doubly linked list）即是这样一个有序节点序列，其中每个节点都有两个指针：left 和 right。left 指针指向左边节点（如果有），right 指针指向右边节点（如果有）。图 3-9 给出了线性表 (1,2,3,4) 的双向链表表示。

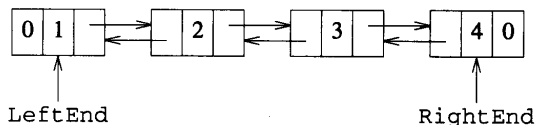


图3-9 一个双向链表

在 C++ 中，可以采用程序 3-21 中给出的类定义来描述双向链表。该定义所定义的函数都是基于链表左部的，如，Find( $k$ ,  $x$ ) 从链表左部开始查找第  $k$  个元素。当然，也可以定义基于链表右部的函数。

程序3-21 双向链表的类定义

```
template <class T>
class DoubleNode {
    friend Double<T>;
private:
    T data;
    DoubleNode<T> *left, *right;
};

template<class T>
class Double {
public:
    Double() {LeftEnd = RightEnd = 0;};
    ~Double();
    int Length() const;
    bool Find(int k, T& x) const;
    int Search(const T& x) const;
    Double<T>& Delete(int k, T& x);
    Double<T>& Insert(int k, const T& x);
    void Output(ostream& out) const;
private:
    DoubleNode<T> *LeftEnd, *RightEnd;
};
```

通过在双向链表的左部和/或右部添加头节点,并且把链表变成循环的链表,可以提高双向链表的性能。在一个非空的双向链表中, LeftEnd->left是一个指向最右边节点的指针(即 RightEnd), RightEnd->right是一个指向最左边节点的指针。所以,可以省去 RightEnd,只需简单地使用变量 LeftEnd来跟踪链表。

### 3.4.8 小结

本节引入了以下重要概念:

- 单向链表 令 $x$ 是一个单向链表。当且仅当 $x.first=0$ 时 $x$ 为空。如果 $x$ 不为空,则 $x.first$ 指向链表的第一个节点。第一个节点指向第二个节点;第二个节点指向第三个节点,如此进行下去。最后一个节点的链指针为0。
- 单向循环链表 它与单向链表的唯一区别是最后一个节点又反过来指向了第一个节点。当循环链表 $x$ 为空时, $x.first=0$ 。
- 头指针 这是在链表中引入的附加节点。利用该节点通常可以使程序设计更简洁,因为这样可以避免把空表作为一种特殊情况来对待。使用头指针时,每个链表(包括空表)都至少包含一个节点(即头指针)。
- 双向链表 双向链表由从左至右按序排列的节点构成。 $right$ 指针用于把节点从左至右链接在一起,最右边节点的 $right$ 指针为0。 $left$ 指针用于把节点从右至左链接在一起,最左边节点的 $left$ 指针为0。
- 双向循环链表 双向循环链表与双向链表的唯一区别在于,最左边节点的 $left$ 指针指向最右边的节点,而最右边节点的 $right$ 指针指向最左边的节点。

### 练习

22. 编写一个复制构造函数  $Chain<T>::Chain(const Chain<T>\& C)$ ,把链表 $C$ 中的元素复制到新的节点中。这个构造函数的复杂性是多少?
23. 编写一个函数,把一个用数组表示的线性表转换成单向链表。要求利用 $LinearList$ 的成员函数 $Find$ 和 $Chain$ 的成员函数 $Insert$ 来实现。该函数的时间复杂性是多少?试测试代码的正确性。
24. 编写一个函数,把一个用单向链表表示的线性表转换成用数组表示的线性表。
  - 1) 首先利用 $Chain$ 的成员函数 $Find$ 和 $LinearList$ 的成员函数 $Insert$ 来实现。该函数的时间复杂性是多少?试测试代码的正确性。
  - 2) 利用链表遍历器来实现。函数的时间复杂性是多少?试用适当的测试数据来测试该函数的正确性。
25. 扩充 $Chain$ 的类定义,把 $LinearList$ 转换成 $Chain$ 以及 $Chain$ 转换成 $LinearList$ 的函数作为成员函数添加到 $Chain$ 的类定义之中。具体任务是编写函数  $FromList(L)$ 和 $ToList(L)$ 。 $FromList(L)$ 把一个线性表 $L$ 转换成单向链表,而 $ToList(L)$ 把一个单向链表转换成线性表 $L$ 。每个函数的时间复杂性分别是多少?试测试代码的正确性。
26. 试比较程序3-12和3-20中 $Search$ 函数的运行性能。分别使用大小为100、1000、10 000和100 000的线性表比较最坏情况下的运行时间及平均运行时间。以表格和图的形式给出所得到的时间。
27. 1) 扩充 $Chain$ 的类定义,增加函数 $Reverse$ ,用于对 $x$ 中的元素反序。要求反序操作就地地进行,不需要分配任何新的节点。

2) 函数的时间复杂性是多少？

3) 通过编译和执行，测试函数的正确性。要求使用自己的测试数据。

28. 完成练习27，区别是Reverse 不作为Chain 的成员函数。要求利用Chain的成员函数来完成反序操作。新的函数将拥有两个参数A和B，A作为输入的链表，B是把A反序后得到的链表。在反序完成时，A变成一个空的链表。

29. 令A 和B 都是Chain 类型

1) 编写一个新的成员函数Alternate，用以创建一个新的线性表C，该表包含了A和B中的所有元素，其中A和B的元素轮流出现，表中的首元素为A中的第一个元素。在轮流排列元素时，如果某个表的元素用完了，则把另一个表的其余元素依次添加在新表的后部。代码的复杂性应与两个输入表的长度呈线性比例关系。

2) 证明代码具有线性复杂性。

3) 通过编译和执行，测试函数的正确性。要求使用自己的测试数据。

30. 扩充Chain的类定义，增加一个与练习29中的Alternate函数相类似的函数Alternate。该函数应利用A和B中的物理节点来建立C。在执行完Alternate之后，A和B均变成空表。

1) 编写出Alternate的实现代码。代码的复杂性应与初始链表的长度呈线性关系。

2) 证明代码具有线性复杂性。

3) 通过编译和执行，测试函数的正确性。要求使用自己的测试数据。

31. 令A 和B 都是Chain 类型，假定A 和B 的元素都是按序排列的（即从左至右按递增量序排列）

1) 编写一个函数Merge，用以创建一个新的有序线性表C，该表中包含了A和B的所有元素。

2) 函数的时间复杂性如何？

3) 通过编译和执行，测试函数的正确性。要求使用自己的测试数据。

32. 重做练习31，要求函数是Chain的一个成员函数，并使用两个输入链表中的物理节点来建立新链表C，在Merge执行完之后，两个输入链表均变成空表。

33. 令C为Chain类型

1) 试编写函数Split，该函数用来创建两个单向链表A和B，A中包含C中所有奇数位置上的元素，B中包含其余的元素。函数不能修改线性表C。

2) 函数的时间复杂性如何？

3) 通过编译和执行，测试函数的正确性。要求使用自己的测试数据。

34. 为Chain类编写一个成员函数Split，该函数与练习33中的Split相类似，区别是本函数破坏输入链表，并采用输入链表中的节点来构造A和B。

35. 设计Circular类。该类的对象是如图3-8所示的循环链表，区别是没有头节点。必须实现为Chain类定义的所有函数（见程序3-8）。每个函数的复杂性分别是多少？测试所编写代码的正确性。

36. 当每个表都有一个头节点时，完成练习35。

37. 用循环链表代替单向链表完成练习27。

38. 用循环链表代替单向链表完成练习29。

39. 用循环链表代替单向链表完成练习31。

40. 用循环链表代替单向链表完成练习33。

41. 令x 指向一个循环链表z 中的任意节点

1) 编写一个函数用来删除节点x 中的元素。提示：由于不知道哪个节点是x 的左边相邻节

点, 因此难以从链表中删除节点  $x$ ; 不过, 为了删除  $x$  中的元素, 可以把  $x$  的数据域用  $x$  的下一个节点  $y$  的数据域来替换, 然后删除  $y$  节点即可。

2) 所编写的函数的时间复杂性如何?

3) 通过编译和执行, 测试函数的正确性。要求使用自己的测试数据。

42. 使用带有头节点的循环链表而不是普通的循环链表完成练习 35。

43. 使用带有头节点的循环链表而不是普通的单向链表完成练习 27。

44. 使用带有头节点的循环链表而不是普通的单向链表完成练习 29, 要求释放额外的头节点。

45. 使用带有头节点的循环链表而不是普通的单向链表完成练习 31, 要求释放额外的头节点。

46. 使用带有头节点的循环链表而不是普通的单向链表完成练习 33, 要求分配一个新的节点, 因为每个新链表都需要有一个头节点。

47. 使用双向链表代替循环链表完成练习 35。

48. 使用双向链表代替普通的单向链表完成练习 27。

49. 使用双向链表代替普通的单向链表完成练习 29。

50. 使用双向链表代替普通的单向链表完成练习 31。

51. 使用双向链表代替普通的单向链表完成练习 33。

52. 使用带有一个头节点的双向循环链表完成练习 35。

53. 使用带有一个头节点的双向循环链表完成练习 27。

54. 使用带有一个头节点的双向循环链表完成练习 29, 要求释放额外的头节点。

55. 使用带有一个头节点的双向循环链表完成练习 31, 要求释放额外的头节点。

56. 使用带有一个头节点的双向循环链表完成练习 33, 要求分配一个新的节点, 因为每个新链表都需要有一个头节点。

57. 为了有效地支持在一个双向链表中进行前移和后移, 需要扩充 `Double` 的类定义 (见程序 3-21), 即增加一个私有成员 `current`, 用它来纪录链表的当前位置, 为此, 需要增加以下共享函数:

1) `ResetLeft`——将 `current` 置为 `LeftEnd`。

2) `ResetRight`——将 `current` 置为 `RightEnd`。

3) `current(x)`——取当前元素至  $x$ 。如果 `current > length`, 函数返回 `false`, 否则返回 `true`。

4) `End`——如果当前的位置恰好指向链表的最后一个元素 (即最右边的元素), 则返回 `true`, 否则返回 `false`。

5) `Front`——如果当前的位置恰好指向链表的第一个元素 (即最左边的元素), 则返回 `true`, 否则返回 `false`。

6) `Next`——移动 `current` 指向链表的下一个元素。如果没有下一个元素, 函数返回 `false`, 否则返回 `true`。

7) `Previous`——移动 `current` 指向链表的前一个元素。如果没有前一个元素, 函数返回 `false`, 否则返回 `true`。

编写以上扩充函数, 使用适当的测试数据测试代码的正确性。

58. 为 `Chain` 增加成员函数 `InsertionSort`, 该函数可利用程序 2-15 给出的插入排序算法对链表中的元素按递增次序进行重新排列。不得创建新的节点或删除老的节点。

1) 程序在最坏情况下的时间复杂性是多少? 如果链表中的元素已经按递增次序排列, 程序需要消耗多长时间?

2) 通过编译和执行, 测试程序的正确性。要求使用自己的测试数据。

59. 分别采用以下排序算法 ( 详见第2章 ) 完成练习58 :

- 1) 冒泡排序。
- 2) 选择排序。
- 3) 计数排序。

## 3.5 间接寻址

### 3.5.1 基本概念

间接寻址 ( indirect addressing ) 是公式化描述和链表描述的组合。采用这种描述方法, 可以保留公式化描述方法的许多优点——可以根据索引在  $O(1)$  的时间内访问每个元素、可采用二叉搜索方法在对数时间内对一个有序表进行搜索等等。与此同时, 也可以获得链表描述方法的重要特色——在诸如插入和删除操作期间不必对元素进行实际的移动。因此, 大多数间接寻址链表操作的时间复杂性都与元素的总数无关。

在间接寻址方式中, 使用一个指针表来跟踪每个元素。可采用一个公式 ( 如公式 ( 3-1 ) ) 来定位每个指针的位置, 以便找到所需要的元素。

元素本身可能存储在动态分配的节点或节点数组之中。图3-10给出了一个采用间接寻址表 table 描述的5元素线性表。其中  $table[i]$  是一个指针, 它指向表中的第  $i+1$  个元素,  $length$  是表的长度。

尽管可以使用公式 ( 3-1 ) 来定位指向表中第  $i$  个元素的指针, 但这个公式本身并不能直接定位第  $i$  个元素。在对表元素的寻址模式中  $table$  提供了一级“间接”引用。

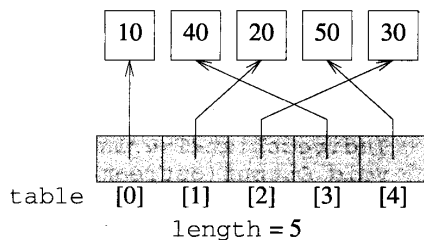


图3-10 间接寻址

如果将图3-10的间接寻址方式与图3-5的链表描述方式进行比较, 可以发现, 二者都使用了指针域 ( 或链接域 )。在链表方式中, 指针位于每个节点中, 而在间接寻址方式中, 指针全放在数组  $table$  之中, 就像是一本书的目录索引一样。为了找到一本书中的某一项内容, 首先需要查目录索引, 索引会告诉我们该项内容在哪里。

当元素存储在动态分配的节点中时, 相应的类定义见程序 3-22。私有成员有:  $table$ 、 $length$  和  $MaxSize$ 。  $table$  是一个指针数组, 用来指向类型为  $T$  的元素;  $MaxSize$  是指针数组的大小, 其省缺值为 10;  $length$  是表的当前长度。

程序3-22 间接寻址表的类定义

```
template<class T>
class IndirectList {
public:
    IndirectList(int MaxListSize = 10);
    ~IndirectList();
    bool IsEmpty() const {return length == 0;}
    int Length() const {return length;}
    bool Find(int k, T& x) const;
    int Search(const T& x) const;
    IndirectList<T>& Delete(int k, T& x);
```

```
IndirectList<T>& Insert(int k, const T& x);  
void Output(ostream& out) const;  
private:  
    T **table; // 一维T类型指针数组  
    int length, MaxSize;  
}
```

### 3.5.2 操作

程序3-23给出了构造函数和析构函数的代码。为了创建一个大小不会超过 20 的空的整数线性表x，可以采用如下的语句：

```
IndirectList<int> x(20);
```

程序3-23 间接寻址的构造函数和析构函数

```
template<class T>  
IndirectList<T>::IndirectList(int MaxListSize)  
{// 构造函数  
    MaxSize = MaxListSize;  
    table = new T *[MaxSize];  
    length = 0;  
}  
template<class T>  
IndirectList<T>::~~IndirectList()  
{// 删除表  
    for (int i = 0; i < length; i++)  
        delete table[i];  
    delete [ ] table;  
}
```

表x的长度可由length给出，在程序3-22中函数Length被定义为一个内联函数，IsEmpty也被定义为一个内联函数。假定  $1 \leq k \leq \text{length}$ ，第k个元素可由table[k-1]之后的下一个指针指出。对x进行搜索可通过依次检查指针 table[0]、table[1]、...所指向的元素。程序3-24给出了函数Find的代码。函数Length、IsEmpty和Find的时间复杂性均为  $O(1)$ 。注意观察这些代码与类LinearList相应代码之间的相似性。

程序3-24 间接寻址的Find函数

```
template<class T>  
bool IndirectList<T>::Find(int k, T& x) const  
{//取第k个元素至x  
    //如果不存在第k个元素，函数返回false，否则返回 true  
    if (k < 1 || k > length) return false; // 不存在第k个元素  
    x = *table[k - 1];  
    return true;  
}
```

为了从图3-10的表中删除第3个元素，需要释放由第3个元素所占用的空间，即把指针table[3:4]移动至table[2:3]，并将length减1。程序3-25是与程序3-4和程序3-14相对应的、进行间



接寻址删除操作的函数。请注意程序 3-4 和程序 3-25 之间的相似性。程序 3-14 和程序 3-25 的时间复杂性均与表的大小无关。不管每个表元素的大小是 10 个字节还是 1000 个字节，这些函数删除一个元素所需要的时间均相同。对于程序 3-4 来说，删除长度为 1000 字节的元素要比删除长度为 10 个字节的元素花费更多的时间，因为它需要移动表元素，每次移动将花费  $\Theta(s)$  的时间，其中  $s$  为一个元素的大小。

程序 3-25 从间接寻址表中删除元素

```
template<class T>
IndirectList<T>& IndirectList<T>::Delete(int k, T& x)
{ // 把第k个元素传送到x，然后删除第k个元素
  // 如果不存在第k个元素，则引发异常 OutOfBounds
  if (Find(k, x)) { // 向前移动指针 k+1, ...
    for (int i = k; i < length; i++)
      table[i-1] = table[i];
    length--;
    return *this;
  }
  else throw OutOfBounds();
}
```

假定要在图 3-10 的表中第 2 和第 3 个元素之间插入一个元素  $x$ ，需要建立如图 3-11 所示的结构。方法是首先将指针  $table[2:4]$  向右移动一个位置，然后在  $table[2]$  中填入一个指向  $y$  的指针。程序 3-26 可在线性表  $x$  的第  $k$  个元素之后插入一个元素，该程序在最坏情况下的时间复杂性与程序 3-15（向链表中插入一个元素）完全相同，都是  $O(\text{length})$ 。而程序 3-5 中基于公式描述的插入函数所拥有的时间复杂性为  $O(s * \text{length})$ ，其中  $s$  是一个类型为  $T$  的元素的大小。

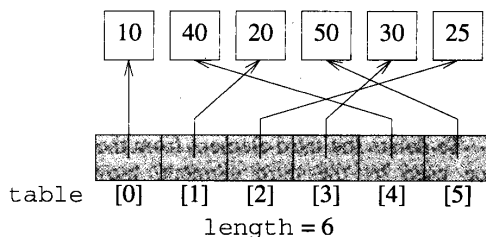


图 3-11 向间接寻址表中插入元素

程序 3-26 向间接寻址表中插入元素

```
template<class T>
IndirectList<T>& IndirectList<T>
::Insert(int k, const T& x)
{ // 在第k个元素之后插入x
  // 如果不存在第k个元素，则引发异常 OutOfBounds
  // 如果没有足够的空间，则传递 NoMem 异常
  if (k < 0 || k > length) throw OutOfBounds();
  if (length == MaxSize) throw NoMem();
  // 向后移动一个位置
  for (int i = length-1; i >= k; i--)
    table[i+1] = table[i];
  table[k] = new T;
  *table[k] = x;
  length++;
}
```



```
return *this;  
}
```

## 练习

60. 编写IndirectList的成员函数Output, 并利用该函数重载操作符<<。测试代码的正确性。
61. 编写IndirectList的成员函数Search, 测试其正确性, 并指出函数的时间复杂性。
62. 设计一个间接寻址的类遍历器, 参照ChainIterator类(见程序3-18)的定义。通过用它从左至右输出一个线性表来测试其正确性。
63. 1) 编写一个折半搜索(见程序2-30)函数, 用来对一个间接寻址表进行搜索。假定对于所有的 $i$   $length-2$ 有  $*table[i] \rightarrow *table[i+1]$ 。函数的时间复杂性应该是  $O(\log(length))$ , 试证明之。  
2) 通过编译和执行, 测试该程序的正确性。要求使用自己的测试数据。  
3) 搜索一个有序的单向链表速度会有多快? 试编写一个具有这种时间复杂性的, 对单向链表进行搜索的函数。
64. 令 $x$ 为IndirectList类型的对象  
1) 编写一个排序函数, 使  $x$ 按递增次序排列, 即对于所有的  $i$   $length-2$ , 有 $*table[i] \rightarrow *table[i+1]$ 。基于插入排序方法(见程序2-15)实现该函数。函数的时间复杂性应该是  $O(length^2)$ , 并且与每个元素的大小无关。试证明之。  
2) 通过编译和执行, 测试该程序的正确性。要求使用自己的测试数据。
65. 分别采用以下排序算法(详见第2章)完成练习64:  
1) 冒泡排序。  
2) 选择排序。  
3) 计数排序。
66. 给定一个类型为 $T$ 的数组 $element[0: length-1]$ 和一个整数数组 $table[0: length-1]$ 。 $table[]$ 是 $[0, 1, \dots, length-1]$ 的一种排列, 使得对于  $0 \leq i < length-2$ 有 $element[table[i]] \rightarrow element[table[i+1]]$ 。  
1) 编写一个函数对 $element[]$ 进行排序, 使得对于所有的 $i$ , 有 $element[i] \rightarrow element[i+1]$ 。函数的时间复杂性应该是  $O(s * length)$ , 其中 $s$ 为每个元素的大小, 函数的空间复杂性应该是  $O(s)$ 。试证明之。  
2) 测试函数的正确性。

## 3.6 模拟指针

在大多数应用中, 可以利用动态分配及C++指针来实现链表和间接寻址表。不过, 有时候采用一个节点数组以及对该数组进行索引的模拟指针(simulated pointer), 可以使设计更方便、更高效。

假定采用一个数组 $node$ , 该数组的每个元素中都包含两个域:  $data$ 和 $link$ 。数组中的节点分别是:  $node[0]$ 、 $node[1]$ 、...、 $node[NumberOfNodes-1]$ 。以下用节点 $i$ 来代表 $node[i]$ 。如果一个单向链表 $c$ 由节点10, 5和24按序构成, 将得到 $c=10$ (指向链表 $c$ 的第一个节点的指针是整数类型),  $node[10].link=5$ (指向第二个节点的指针),  $node[5].link=24$ (指向下一个节点的指针),  $node[24].link=-1$ (表示节点24是链表中的最后一个节点)。在绘制链表时, 可以把每个链接指针

画成一个箭头（如图3-12所示），与使用C++指针的时候一样。

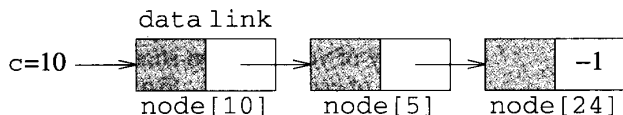


图3-12 采用模拟指针的链表

为了实现指针的模拟，需要设计一个过程来分配和释放一个节点。当前未被使用的节点将被放入一个存储池（storage pool）之中。开始时，存储池中包含了所有节点 node[0: NumberOfNodes-1]。Allocate从存储池中取出节点，每次取出一个。Deallocate则将节点放入存储池中，每次放入一个。因此，Allocate和Deallocate分别对存储池执行插入和删除操作，等价于C++函数delete和new。如果存储池是一个节点链表（如图3-13所示），这两个函数可以高效地执行。用作存储池的链表被称之为可用空间表（available space list），其中包含了当前未使用的所有节点。first是一个类型为int的变量，它指向可用空间表中的第一个节点。添加和删除操作都是在可用空间表的前部进行的。

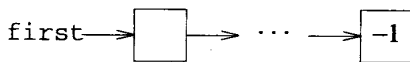


图3-13 可用空间表

为了实现一个模拟指针系统，定义了SimNode类和SimSpace类，见程序3-27。

程序3-27 模拟指针的类定义

```
template <class T>
class SimNode {
    friend SimSpace<T>;
private:
    T data;
    int link;
};

template <class T>
class SimSpace {
public:
    SimSpace (int MaxSpaceSize=100);
    ~SimSpace() {delete [] node;}
    int Allocate(); //分配一个节点
    void Deallocate (int& i); //释放节点i
private:
    int NumberOfNodes, first;
    SimNode<T> *node; //节点数组
};
```

### 3.6.1 SimSpace的操作

由于所有节点初始时都是自由的，因此在刚被创建的时候，可用空间表中包含NumberOfNodes个节点。程序3-28用来对可用空间表进行初始化。程序3-29和程序3-30分别实现Allocate和Deallocate操作。

程序3-28 初始化可用空间表

```
template<class T>
SimSpace<T>::SimSpace(int MaxSpaceSize)
{ //构造函数
    NumberOfNodes = MaxSpaceSize;
    node = new SimNode<T> [NumberOfNodes];
    //初始化可用空间表
    //创建一个节点链表
    for (int i = 0; i < NumberOfNodes-1; i++)
        node[i].link = i+1;
    //链表的最后一个节点
    node[NumberOfNodes-1].link = -1;
    //链表的第一个节点
    first = 0;
}
```

程序3-29 使用模拟指针分配一个节点

```
template<class T>
int SimSpace<T>::Allocate()
{ // 分配一个自由节点
    if (first == -1) throw NoMem();
    int i = first; //分配第一个节点
    first = node[i].link; //first指向下一个自由节点
    return i;
}
```

程序3-30 使用模拟指针释放一个节点

```
template<class T>
void SimSpace<T>::Deallocate(int& i)
{ // 释放节点i.
    //使 i 成为可用空间表的第一个节点
    node[i].link = first;
    first = i;
    i = -1;
}
```

以上三个函数的时间复杂性分别为  $\Theta(\text{NumberOfNodes})$ ,  $\Theta(1)$  和  $\Theta(1)$ 。通过使用两个可用空间表,可以减少构造函数(见程序3-28)的运行时间,其中第一个表包含所有尚未被使用的自由节点,第二个表包含所有已被至少使用过一次的自由节点。每当一个节点被释放时,被放入第二个表中。当需要一个新节点时,如果第二个表非空,则从该表中取出一个节点,否则从第一个表中取出一个节点。令  $\text{first1}$  和  $\text{first2}$  分别指向第一个表和第二个表的首节点。基于上述分配节点的方式,第一个表中的元素为  $\text{node}[i]$ , 其中  $\text{first1} \leq i < \text{NumberOfNodes}$ 。释放一个节点的代码与程序3-30的唯一区别在于将所有  $\text{first}$  变量均替换成  $\text{first2}$ 。新的构造函数和分配函数分别见程序3-31和程序3-32。为了使新函数能正常工作,需要把整型变量  $\text{first1}$  和  $\text{first2}$  设置为  $\text{SimSpace}$  的私有成员变量。

程序3-31 使用两个可用空间表的构造函数

```
template<class T>
SimSpace<T>::SimSpace(int MaxSpaceSize)
{ // 使用两个可用空间表的构造函数
    NumberOfNodes = MaxSpaceSize;
    node = new SimNode<T> [NumberOfNodes];
    //初始化可用空间表
    first1 = 0;
    first2 = -1;
}
```

程序3-32 使用两个可用空间表的 Allocate函数

```
template<class T>
int SimSpace<T>::Allocate()
{//分配一个自由节点
    if (first2 == -1) {// 第2个表为空
        if (first1 == NumberOfNodes) throw NoMem();
        return first1++;}
    //分配链表中的第一个节点
    int i = first2;
    first2 = node[i].link;
    return i;
}
```

我们希望在大多数应用中，由于使用了两个可用空间表，程序 3-31和程序3-32应该比只使用一个可用空间表的程序提供更好的性能。为此做以下的考察：

- 程序3-32所花费的时间与程序 3-29所花费的时间相同，除非节点是从第一个表中取出的。这种例外至多发生 NumberOfNodes 次。在这种例外情况下额外花费的时间将抵销了初始化所节省出的时间。事实上，需要的节点数通常少于 NumberOfNodes 个（尤其是在调试程序以及解决实例特征变化较大的问题的时候），因此两个可用空间表模式将运行得更快。

- 在一个交互式的环境中减少初始化所需要的时间是很受欢迎的，因为程序的启动时间将会大大减少。

- 在只使用一个可用空间表时，对于所建立的单向链表，除了最后一个节点外，没有必要明确指定节点的链接指针，因为节点中已经体现了正确的链接值（如图 3-13所示）。这种优点也可以引入使用两个可用空间表的模式中，方法是编写一个 Get(n)函数，它产生一个有 n个节点的单向链表。仅当从第一个表中取出一个节点时，Get(n)函数才明确地设置链接域的值。

- 采用可用空间表模式分解一个链表将比采用 C++ 指针更高效。例如，如果一个单向链表的首部和尾部分别为 f 和 e，可以采用如下语句来释放链表中的所有节点：

```
node[e].link = first; first = f;
```

- 如果 c 是一个循环链表，则采用程序 3-33释放表中所有节点需要的时间为  $\Theta(1)$ 。图3-14给出了链接指针所发生的变化。

程序3-33 释放一个循环链表

```

template<class T>
void SimSpace<T>::DeallocateCircular(int& c)
{
    // 释放一个循环链表c
    if(c != -1) {
        int next = node[c].link;
        node[c].link = first;
        first = next;
        c = -1;
    }
}

```

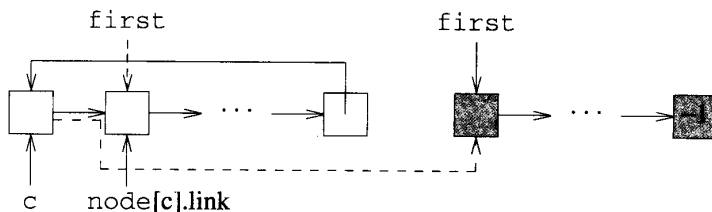


图3-14 释放一个循环链表

### 3.6.2 采用模拟指针的链表

可以使用模拟空间S来定义一个链表类（见程序3-34）。S被说明为一个static成员，目的是使所有类型为T的模拟链表共享相同的模拟空间。程序3-35至程序3-38给出了除Search和Output之外的各共享函数的代码。这些代码假定SimChain已经被说明为SimNode和SimSpace的友元。注意观察这些代码与Chain各相应成员函数之间的相似性。程序3-39给出了使用模拟链表的示例程序。在该程序中，simul.h和schain.h分别包含了SimSpace类和SimChain类的代码。

程序3-34 模拟链表的类定义

```

template<class T>
class SimChain {
public:
    SimChain() {first = -1;}
    ~SimChain() {Destroy();}
    void Destroy(); // 使表为空
    int Length() const;
    bool Find(int k, T& x) const;
    int Search(const T& x) const;
    SimChain<T>& Delete(int k, T& x);
    SimChain<T>& Insert(int k, const T& x);
    void Output(ostream& out) const;
private:
    int first; // 第一个节点的索引
    static SimSpace<T> S;
};

```

程序3-35 模拟指针的构造函数和Length函数

```
template<class T>
void SimChain<T>::Destroy()
{// 释放链表节点
    int next;
    while (first != -1) {
        next = S.node[first].link;
        S.Deallocate(first);
        first = next;}
}
template<class T>
int SimChain<T>::Length() const
{// 返回链表的长度
    int current = first ; //链节点的当前位置
    len = 0;      //元素计数
    while (current != -1) {
        current = S.node[current].link;
        len++;}
    return len;
}
```

程序3-36 模拟指针的Find函数

```
template<class T>
bool SimChain<T>::Find(int k, T& x) const
{// 取第k个元素至x
    //如果不存在第k个元素，函数返回 false，否则返回 true
    if (k < 1) return false;
    int current = first, // 链节点的当前位置
        index = 1;      //当前节点的索引
    //移动current至第k个节点
    while (index < k && current != -1) {
        current = S.node[current].link;
        index++;}
    //验证是否到达了第k个节点
    if (current != -1) {x = S.node[current].data; return true;}
    return false; // 不存在第k个元素
}
```

程序3-37 模拟指针的Delete函数

```
template<class T>
SimChain<T>& SimChain<T>::Delete(int k, T& x)
{//把第k个元素取至x，然后删除第k个元素
    //如果不存在第k个元素，则引发异常 OutOfBounds
    if (k < 1 || first == -1)
        throw OutOfBounds(); // 不存在第k个元素
    // p 最终将指向第k个节点
```

```

int p = first;
//将p移动至第k个节点，并从链表中删除该节点
if (k == 1) // p已经指向第k个节点
    first = S.node[first].link; // 从链表中删除
else { // 使用q指向第k-1个元素
    int q = first;
    for (int index = 1; index < k - 1 && q != -1; index++)
        q = S.node[q].link;
// 验证第k个元素的存在性
if (q == -1 || S.node[q].link == -1)
    throw OutOfBounds(); // 不存在第k个元素
//使p指向第k个元素
p = S.node[q].link;
//从链表中删除第k个元素
S.node[q].link = S.node[p].link;
}
//保存第k个元素并释放节点p
x = S.node[p].data;
S.Deallocate(p);
return *this;
}

```

程序3-38 模拟指针的Insert函数

```

template<class T>
SimChain<T>& SimChain<T>::Insert(int k, const T& x)
{
//在第k个元素之后插入x
//如果不存在第k个元素，则引发异常 OutOfBounds
//如果没有足够的空间，则传递 NoMem异常
if (k < 0) throw OutOfBounds();
//定义一个指针 p，p最终将指向第k个节点
int p = first;
//将p移向第k个节点
for (int index = 1; index < k && p != -1; index++)
    p = S.node[p].link;
// 验证第k个节点的存在性
if (k > 0 && p == -1)
    throw OutOfBounds();
// 为插入操作分配一个新节点
int y = S.Allocate();
S.node[y].data = x;
//向链表中插入新节点
// 首先检查新节点是否要插到链表的首部
if (k) { //在p之后插入
    S.node[y].link = S.node[p].link; S.node[p].link = y;
}
else { // 作为链表首节点
    S.node[y].link = first; first = y;
}
return *this;
}

```



程序3-39 使用模拟链表

---

```

#include <iostream.h>
#include "schain.h"
SimSpace<int> SimChain<int>::S;
void main(void)
{
    int x;
    SimChain<int> c;
    cout << "Chain length is" << c.Length() << endl;
    c.Insert(0, 2).Insert(1, 6);
    cout << "Chain length is" << c.Length() << endl;
    c.Find(1, x);
    cout << "First element is" << x << endl;
    c.Delete(1, x);
    cout << "Deleted" << x << endl;
    cout << "New length is" << c.Length() << endl;
    cout << "Position of 2 is" << c.Search(2) << endl;
    cout << "Position of 6 is" << c.Search(6) << endl;
    c.Insert(0, 9).Insert(1, 8).Insert(2, 7);
    cout << "Current chain is" << c << endl;
    cout << "Its length is" << c.Length() << endl;
}

```

---

## 练习

67. 为SimChain类设计一个类遍历器SimIterator。请参考程序3-18中关于Chains类遍历器的定义。SimIterator中应包含与ChainIterator相同的成员函数。编写并测试代码。

68. 1) 修改SimSpace类, 使得Allocate返回一个指向node[i]的指针, 而不是返回索引值i。类似地, 修改Deallocate函数, 使它的输入参数为欲释放节点的指针。

2) 采用1) 中的SimSpace代码重新编写SimChain的实现代码。请留意新代码与Chain的代码之间的相似性。

69. 1) 修改SimNode类的定义, 为它添加一个类型为SimSpace<T>的静态成员S。这样, 所有类型为SimSpace<T>的节点都可以共享同样的模拟空间。重载函数new和delete, 以便从模拟空间S中取得节点SimNodes或将SimNodes节点送回模拟空间S中。

2) 假定SimSpace是按照练习68的要求实现的, SimNode是按照1) 中要求实现的。修改Chain类的代码(见程序3-8), 使得它能够用SimNodes代替ChainNodes进行工作。测试代码, 并测量运行时间, 以确定哪个版本的Chain更快。

70. 假定一个链表是采用模拟指针进行描述的, 节点的类型为SimNode

1) 编写一个程序, 该程序使用插入排序算法对链表中的节点进行重新排序, 要求按照data域的递增次序进行排列。

2) 代码的时间复杂性是多少? 如果不是 $O(n^2)$ , 请重写代码以使其具有这样的复杂性, 其中 $n$ 为链表长度。

3) 测试代码的正确性。

71. 使用选择排序算法完成练习70。

72. 使用冒泡排序算法完成练习70。

73. 使用计数排序算法完成练习70。

74. 对new和delete的调用通常都要耗费很多时间，为此，可以使用自行编写的释放函数（该函数能将删除的节点放入自由节点表中）来替换 delete函数，以便提高代码的运行效率。为了替换new，可以自行编写一个分配函数，每当自由节点链表为空时该函数才会去调用 new。修改Chain类（见程序3-8）以实现上述思想。要求编写如上所述的分配节点和释放节点函数，并对自由节点链表进行初始化。试比较两种 Chain版本的执行时间，并评价新方法的优缺点。

75. 考察按如下方式定义的XOR操作（异或操作，也可以记为 + ）：

$$i \oplus j = \begin{cases} 0 & i=j \\ 1 & i \neq j \end{cases}$$

两个二进制串*i*和*j*的XOR操作结果由*i*和*j*各相应位的XOR结果组成。例如，如果*i*=10110且*j*=01100，则*i*XOR*j*=*i* + *j*=11010。注意有：

$$\begin{aligned} a \oplus (a \oplus b) &= (a \oplus a) \oplus b = b \\ (a \oplus b) \oplus b &= a \oplus (b \oplus b) = a \end{aligned}$$

以上规律提供了一种节省双向链表左、右链接指针所需存储空间的结构。假定可用的节点都放在数组node之中，节点的索引号分别为1, 2, ...。因此，node[0]尚未使用。现在可以用0而不是-1来表示NULL指针。每个节点都有两个域：data和link。如果l是节点x的左指针，r是其右指针。对于最左边的节点，l=0，而对于最右边的节点，r=0。令(l, r)是按上述方法实现的双向链表，l指向链表最左边的节点，r指向最右边的节点。

- 1) 编写一个函数从左至右遍历双向链表(l, r)，并输出每个节点data域的内容。
- 2) 编写一个函数从右至左遍历双向链表(l, r)，并输出每个节点data域的内容。
- 3) 测试代码的正确性。

### 3.7 描述方法的比较

在图3-15中，分别给出了使用本章所介绍的四种数据描述方法执行各种链表操作所需要的时间复杂性。在表中，*s*和*n*分别表示sizeof(T)和链表长度。由于采用C++指针和采用模拟指针完成这些操作所需的时间复杂性完全相同，因此表中在把这两种情形的时间复杂性合并在同一行中进行描述。

描述方法	操作		
	查找第 <i>k</i> 个元素	删除第 <i>k</i> 个元素	在第 <i>k</i> 个元素后插入
公式化描述	$\Theta(1)$	$O((n-k)s)$	$O((n-k)s)$
以公式(3-1)为例			
链表描述	$O(k)$	$O(k)$	$O(k+s)$
C++及(模拟指针)			
间接寻址	$\Theta(l)$	$O(n-k)$	$O(n-k)$

图3-15 四种描述方法的比较

使用间接寻址与使用链表描述所需要的空间大致相同，二者都比使用公式化描述所需要的

空间更多。不管是使用链表描述还是间接寻址，执行链表的插入和删除操作所需要的时间复杂性均与每个链表元素本身的大小无关。然而，在使用公式化描述时，插入和删除操作的复杂性与元素本身的大小成线性关系。所以，如果链表元素的大小  $s$  很大，那么使用链表描述和间接寻址将更适合于需要大量插入和删除操作的应用。

在使用公式化描述和间接寻址时，确定表的长度以及访问表中第  $k$  个元素所需要的时间复杂性均为  $\Theta(1)$ 。而在使用链表描述时，这些操作的时间复杂性分别为  $\Theta(\text{length})$  和  $O(k)$ ，所以链表描述不适合于这两种操作占优势的应用。

基于以上的讨论可以发现，间接寻址比较适合这样的应用：表元素本身很大，较频繁地进行插入、删除操作以及确定表的长度、访问第  $k$  个元素。同时，如果线性表本身已经按序排列，那么使用公式化描述或间接寻址进行搜索所需要时间均为  $O(\log n)$ ，而使用链表描述时，所需要的时间为  $O(n)$ 。

## 3.8 应用

### 3.8.1 箱子排序

假定一个链表中包含了一个班级内所有学生的信息，每个节点中含有这样的域：学生姓名、社会保险号码、每次作业和考试的分数以及所有作业和考试的加权总分。假定所有的分数均为 0~100 范围内的整数。如果采用第 2 章中所给出的任一种排序算法对表中的学生按分数进行排序，所需要花费的时间均为  $O(n^2)$ ，其中  $n$  为班级中的学生总数。一种更快的排序方法为箱子排序 (bin sort)。在箱子排序过程中，节点首先被放入箱子之中，具有相同分数的节点都放在同一个箱子中，然后通过把箱子链接起来就可以创建一个有序的链表。

图 3-16a 给出了一个箱子排序的例子，图中的链表含有 10 个节点。该图仅列出了每个节点的姓名域和分数域。第一个域为姓名，第二个域为分数。为简便起见，假定每个姓名为一个字符，分数则介于 0 到 5 之间。需要六只箱子来分别存放具有 0~5 之间某种分数的节点。图 3-16b 给出了 10 个节点按分数分布于各个箱子中的情形。通过沿链表逐个检查每个节点，即可得到这

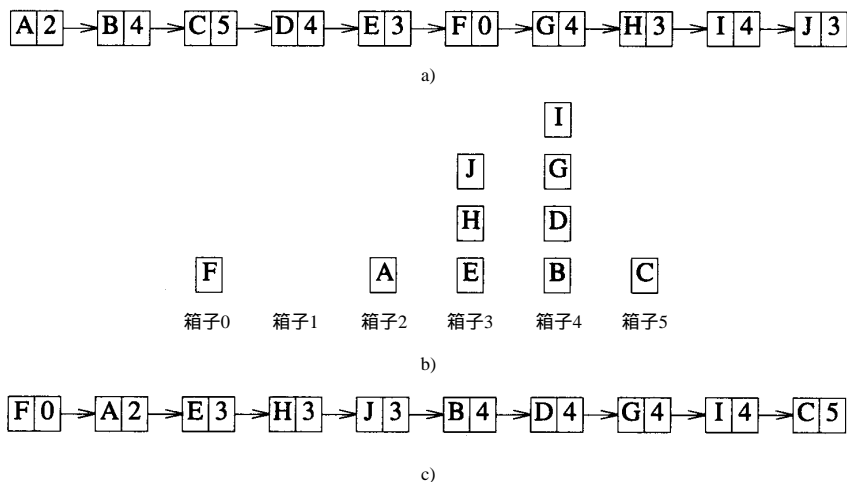


图3-16 箱子排序举例

a) 输入链表 b) 箱子中的节点 c) 排序后的链表

样的分布。当检查某个节点时，该节点被放入与它的分数相对应的那个箱子之中。所以第一个节点被放入2号箱子，第二个节点被放入4号箱子，依此类推。现在，如果从0号箱子开始收集节点，将得到一个如图3-16所示的有序链表。

怎样实现箱子呢？注意到每个箱子都是一个由节点组成的线性表。箱子中的节点数目介于0到n之间。一种简单的方法就是把每个箱子都描述成一个链表。在进行节点分配之前，所有的箱子都是空的。

对于箱子排序，需要能够：1)从欲排序链表的首部开始，逐个删除每个节点，并把所删除的节点放入适当的箱子中（即相应的链表中）；2)收集并链接每个箱子中的节点，产生一个排序的链表。如果所输入的链表为Chain类型（见程序3-8），那么可以：1)连续地删除链表首元素并将其插入到相应箱子链表的首部；2)逐个删除每个箱子中的元素（从最后一个箱子开始）并将其插入到一个初始为空的链表的首部。

链表节点的数据域应该是Node类型（见程序3-40）。操作符!=和<<已经被重载，因为Chain类需要使用这两个操作符。

程序3-40 一种用于箱子排序的节点类

```
class Node {
    friend ostream& operator<<(ostream&, const Node &);
public:
    int operator !=(Node x) const
    {return (score!= x.score);}
private:
    int score;
    char *name;
};
ostream& operator<<(ostream& out, const Node& x)
{out << x.score << ' '; return out;}
```

另一种可选的重载方式是提供一种从Node类型到数字类型的转换，因为数字类型可用于比较和输出。例如，可以重载类型转换操作符int()，见程序3-41。那些在Node类型中未定义的算术和逻辑操作符（比如+，/，<=，!=和输出操作符<<等）现在可以通过首先执行一个类型转换（从定义这些操作符的类型向int类型）而成功地完成相应的操作。这种解决方法与前面介绍的重载操作符!=和<<相比可能会更通用，因为如果这样做，即使扩充了Chain类，为该类增添了对T->data进行其他操作的函数，这些类型转换程序仍可以正常工作。

程序3-41 另一种处理操作符重载的方法

```
class Node {
public:
    //重载类型转换操作符
    operator int() const (return score;)
private:
    int score;
    char *name;
};
```

上述两种重载方法可以联合使用，以便于仅当缺少类型转换操作符而导致失败时才去执行

int类型转换。因此，可以采用程序3-42中的定义。其中，向int类型的转换仅针对除!=和<<之外的操作符。

程序3-42 又一种处理操作符重载的方法

---

```
class Node {
    friend ostream& operator<<(ostream&, const Node &);
public:
    int operator !=(Node x) const
    {return (score != x.score
        || name[0] != x.name[0]);}
    operator int() const {return score;}
private:
    int score;
    char *name;
};
ostream& operator<<(ostream& out, const Node& x)
{
    out << x.score << ' ' << x.name[0] << ' ';
    return out;
}
```

---

程序3-43给出了箱子排序函数的代码，该函数假定 BinSort是Node的一个友元。如果没有足够的空间来创建箱子，该函数允许传递 NoMem异常。在两个for循环中执行的每一次插入和删除操作所需要的时间均为  $\Theta(1)$ ，因此第一个for循环的复杂度为  $\Theta(n)$ ，其中  $n$  为输入链表的长度，第二个for循环的复杂度为  $(n+range)$ ，因此函数 BinSort总的复杂度为  $(n+range)$  (如果成功的话)。

程序3-43 箱子排序

---

```
void BinSort(Chain<Node>& X, int range)
{
    // 按分数排序
    int len = X.Length();
    Node x;
    Chain<Node> *bin;
    bin = new Chain<Node> [range + 1];
    //分配到每个箱子中
    for (int i = 1; i <= len; i++) {
        X.Delete(1,x);
        bin[x.score].Insert(0,x);
    }
    //从箱子中收集各元素
    for (int j = range; j >= 0; j--)
        while (!bin[j].IsEmpty()) {
            bin[j].Delete(1,x);
            X.Insert(0,x);
        }
    delete [ ] bin;
}
```

---

### 1. 把BinSort定义成Chain类的成员

细心的读者可能已经注意到，如果把 BinSort 定义成 Chain 的一个成员函数，可以大大简化 BinSort 函数。当一个元素是链表中的成员并被放入箱子中时，这种方法能够让我们使用相同的物理节点，而且这种方法还可以消除所有对 new 和 delete 的调用（与 bin 相关的那些调用除外）。此外，通过跟踪每个箱子链表的首节点和尾节点，可以链接处于“收集状态”的箱子链表。见程序 3-44。

程序 3-44 Binsort 作为 Chain 类的成员

```
template<class T>
void Chain<T>::BinSort(int range)
{
    // 按分数排序
    int b; // 箱子索引号
    ChainNode<T> **bottom, **top;
    // 箱子初始化
    bottom = new ChainNode<T>* [range + 1];
    top = new ChainNode<T>* [range + 1];
    for (b = 0; b <= range; b++)
        bottom[b] = 0;
    // 把节点分配到各箱子中
    for (; first; first = first->link) { // 添加到箱子中
        b = first->data;
        if (bottom[b]) { // 箱子非空
            top[b]->link = first;
            top[b] = first;
        }
        else { // 箱子为空
            bottom[b] = top[b] = first;
        }
    }
    // 收集各箱子中的元素，产生一个排序链表
    ChainNode<T> *y = 0;
    for (b = 0; b <= range; b++)
        if (bottom[b]) { // 箱子非空
            if (y) { // 不是第一个非空的箱子
                y->link = bottom[b];
            }
            else { // 第一个非空的箱子
                first = bottom[b];
                y = top[b];
            }
            if (y) y->link = 0;
        }
    delete [] bottom;
    delete [] top;
}
```

对应于每个箱子的链表都是以箱子的底部节点作为首节点，其他节点依次排列直至箱子的顶部节点。每个箱子链表都有两个指针：bottom 和 top，它们均指向该链表。bottom[b] 指向箱子 b 的底部节点，而 top[b] 指向箱子 b 的顶部节点。所有空箱子的初始结构可以定义为 bottom[b]=0（见程序 3-44 的第一个 for 循环）。在检查节点时，每个节点均被添加到相应箱子的顶部（见程序 3-44 的第二个 for 循环）。在第二个 for 循环的代码中，为了能返回 score 域，假定已经定义了从 Node 到 int 类型的转换。第三个 for 循环从 0 号箱子开始进行检查，依次把每个非空的

箱子链接在一起以产生一个有序的链表。

至于BinSort的时间复杂性，可以看到，第一和第三个 for循环所需要的时间为  $\Theta(\text{range})$ ，第二个for 循环所需要的时间为  $\Theta(n)$ ，因此总的时间复杂性为  $\Theta(n+\text{range})$ 。

可以注意到BinSort函数并未改变具有同样分数的节点之间的相对次序。例如，假定在输入链中E、G和H的分数均为3，E出现在G之前，G出现在H之前，那么，在排序后的链表中，E仍出现在G之前，G也仍然出现在H之前。在有些排序应用中，要求排序算法不得改变同值元素之间的相对次序。如果一个排序算法能够保持同值元素之间的相对次序，则该算法被称之为稳定排序（stable sort）。

## 2. 概括

假定Node的每个元素都含有 exam1, exam2, exam3以及其他附加的域。在某些程序中，我们可能希望按exam1域进行排序，之后某个时刻可能又希望按 exam3域进行排序，再之后的某个阶段又可能需要按照 exam1+exam2+exam3进行排序。如果我们定义了数据类型 Node1, Node2和Node3，那么可以利用程序 3-44中的代码来执行这三种排序。在 Node1中，可将函数 int()定义为返回exam1域的值，在Node2中则定义为返回exam2域的值，在Node3中定义为返回 exam1+ exam2+exam3的值。在调用BinSort之前，必须把欲排序的数据复制到类型为 Node1或Node2或Node3（取决于所使用的排序值）的链表之中。

为了避免复制链表元素所带来的额外开销，可以为 BinSort增加一个附加的参数 value，并使该函数返回排序所使用的值。其语法如下：

```
void Chain<T>::BinSort(int range, int(*value)(T& x))
```

该语句表明函数Chain<T>::BinSort带有两个参数，而不返回任何值。第一个参数 range的类型为int，第二个参数 value是一个函数的名字，该函数带有一个类型为 T&的参数x并返回一个int值。

当BinSort按照上述形式定义时，程序 3-44中的语句

```
j=first->data;
```

需要被改写为：

```
j=value(first->data);
```

至此，可以采用类似程序3-45的代码来进行排序。

程序3-45 按不同的域进行排序

```
inline int F1(Node& x) {return x.exam1;}
inline int F2(Node& x) {return x.exam2;}
inline int F3(Node& x)
{return x.exam1 + x.exam2 + x.exam3;}
void main(void)
{
    Node x;
    Chain<Node> L;
    randomize();
    for (int i = 1; i <= 20; i++) {
        x.exam1= i/2;
        x.exam2= 20 - i;
        x.exam3= random(100);
        x.name = i;
        L.Insert(0,x);}
}
```



```

L. BinSort(10, F1);
cout << "Sort on exam 1" << endl;
cout << L << endl;
L. BinSort(20, F2);
cout << "Sort on exam 2" << endl;
cout << L << endl;
L. BinSort(130, F3);
cout << "Sort on sum of exams" << endl;
cout << L << endl;
}

```

### 3.8.2 基数排序

可以扩充3.8.1节的箱子排序方法，使其在  $\Theta(n)$  时间内对范围在  $0 \sim n^c - 1$  之间的  $n$  个整数进行排序，其中  $c$  是一个常量。注意，如果用  $\text{range} = n^c$  来调用函数 `BinSort`，则排序的复杂性将变成  $\Theta(n + \text{range}) = \Theta(nc)$ 。对于 `BinSort` 来说，一种替代的方法是，不直接对这些数进行排序，而是采用一些基数  $r$  来分解这些数。例如，十进制数 928 可以按照基数 10 分解为数字 9, 2 和 8（即  $928 = 9 \times 10^2 + 2 \times 10^1 + 8 \times 10^0$ ）。最高位是 9，最低位是 8。用基数 10 来分解 3725 可得到 3, 7, 2 和 5，而利用基数 60 来进行分解则可以得到 1, 2 和 5（即  $(3725)_{10} = (125)_{60}$ ）。在基数排序（radix sort）中，把数按照某种基数分解为数字，然后对数字进行排序。

例3-1 假定对范围在 0~999 之间的 10 个整数进行排序。如果使用  $\text{range} = 1000$  来调用 `BinSort`，那么箱子的初始化将需要 1000 个执行步，节点分配需要 10 个执行步，从箱子中收集节点需要 1000 个执行步，总的执行步数为 2010。另一种方法是：

1) 用 `BinSort` 根据数的最低位数字对 10 个数进行排序。由于每个数字的范围为 0~9，因此  $\text{range} = 10$ 。图3-17a 给出了具有 10 个数的链表，图3-17b 给出了按最低位数字排序后的链表。

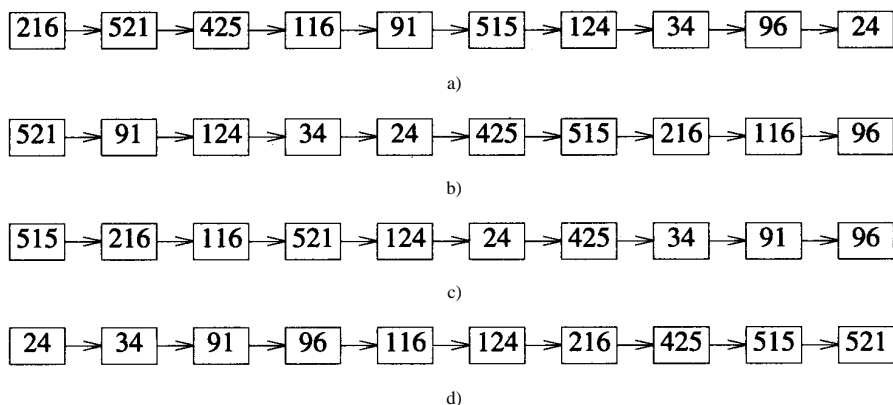


图3-17 用  $r=10$  和  $d=3$  进行基数排序

a) 输入链表 b) 按最后一位数字排序后的链表 c) 按倒数第二位数字排序后的链表 d) 按最高位数字排序后的链表

2) 用箱子排序算法对 1) 中所得到的链表按次低位数字进行排序。同样，有  $\text{range} = 10$ 。由于箱子排序是稳定排序，次低位数字相同的节点，其相对次序保持不变（与按最低位数字排序时

所得到的次序相同)。因此,现在链表是按照最后两位数字进行排序的。图 3-17c 给出了相应的排序结果。

3) 用箱子排序算法对 2) 中所得到的链表按第三位(最高位)数字进行排序。(如果一个数仅包含两位数字,则其第三位数字为 0)。由于按第三位数字排序是稳定排序,所以第三位数字相同的节点,其相对次序保持不变(与按最后两位数字排序时所得到的次序相同)。因此,现在链表是按照后三位数字进行排序的。图 3-17d 给出了相应的排序结果。

上述排序模式是基数为 10 的排序,被排序的数被分解为相应的十进制数字,排序是针对这些数字进行的。由于每个数都至少应有三位数字,所以要进行三次排序,每次排序都使用  $\text{range}=10$  的箱子排序过程来完成。在每次的箱子排序过程中,需要 10 个执行步来对箱子进行初始化,10 个执行步用来把数分配至相应的箱子节点,10 个执行步用来收集箱子节点。总的执行步数为 90,比使用  $\text{range}=1000$  进行 10 个数的箱子排序要少得多。单个箱子排序模式实际上等价于  $r=1000$  的基数排序。

例 3-2 假定对 1000 个范围在  $0 \sim 10^6-1$  之间的整数进行排序,使用基数  $r=10^6$  的排序方法(即直接使用 BinSort 排序函数)需要  $10^6$  执行步对箱子初始化,1000 个执行步分配箱子节点,另外  $10^6$  执行步收集箱子节点,因此总的执行步数为 2 001 000。与此相应地,对于  $r=1000$  的排序,其过程如下:

1) 采用每个数的最低三位数字进行排序,令  $\text{range}=1000$ 。

2) 对 1 中得到的结果再利用每个数的倒数次三位(即倒数第四到六位)数字进行排序。

上述每次排序都需要 3000 个执行步,所以排序完成时共需要 6000 个执行步。若使用  $r=100$ ,则需使用三次箱子排序过程依次对每两位数字进行排序,每次箱子排序需要 1200 个执行步,总的执行步数为 3600。如果使用  $r=10$ ,则要进行六次箱子排序,每次针对一位数字,总的执行步数为  $6(10+1000+10)=6120$ 。因此,对于本例,采用基数  $r=100$  的排序效率最高。

为了实现例 3-1 和例 3-2 的基数排序,需要按给定的基数对数进行分解。可以采用除法和取模运算来完成这种分解。如果采用基数 10 来进行分解,那么可以按照如下表达式来得到每位数字(从最低位到最高位):

$$x \% 10; (x \% 100) / 10; (x \% 1000) / 100; \dots$$

若  $r=100$ ,则相应的分解式为:

$$x \% 100; (x \% 10000) / 100; (x \% 1000000) / 10000; \dots$$

对于一般的基数  $r$ ,相应的分解式为:

$$x \% r; (x \% r^2) / r; (x \% r^3) / r^2; \dots$$

当使用基数  $r=n$  对  $n$  个介于  $0 \sim n^c-1$  范围内的整数进行分解时,每个数将可以分解出  $c$  个数字。因此,可以采用  $c$  次箱子排序,每次排序时取  $\text{range}=n$ 。整个排序所需要的时间为  $\Theta(cn)=\Theta(n)$ (因为  $c$  是一个常量)。

### 3.8.3 等价类

#### 1. 定义和动机

假定有一个具有  $n$  个元素的集合  $U=\{1, 2, \dots, n\}$ ,另有一个具有  $r$  个关系的集合  $R=\{(i_1, j_1), (i_2, j_2), \dots, (i_r, j_r)\}$ 。关系  $R$  是一个等价关系(equivalence relation),当且仅当如下条件为真时成立:

- 对于所有的  $a$ , 有  $(a, a) \in R$  时 (即关系是反身的)。
- 当且仅当  $(b, a) \in R$  时  $(a, b) \in R$  (即关系是对称的)。
- 若  $(a, b) \in R$  且  $(b, c) \in R$ , 则有  $(a, c) \in R$  (即关系是传递的)。

在给出等价关系  $R$  时, 我们通常会忽略其中的某些关系, 这些关系可以利用等价关系的反身、对称和传递属性来获得。

例3-3 假定  $n=14$ ,  $R=\{(1,11), (7,11), (2,12), (12,8), (11,12), (3,13), (4,13), (13,14), (14,9), (5,14), (6,10)\}$ 。我们忽略了所有形如  $(a,a)$  的关系, 因为按照反身属性, 这些关系是隐含的。同样也忽略了所有的对称关系。比如  $(1,11) \in R$ , 按对称属性应有  $(11,1) \in R$ 。其他被忽略的关系是由传递属性可以得到的属性。例如根据  $(7,11)$  和  $(11,12)$ , 应有  $(7,12) \in R$ 。

如果  $(a,b) \in R$ , 则元素  $a$  和  $b$  是等价的。等价类 (equivalence class) 是指相互等价的元素的最大集合。“最大”意味着不存在类以外的元素, 与类内部的元素等价。

例3-4 考察例3-3中的等价关系。由于元素 1 与 11, 11 与 12 是等价的, 因此, 元素 1, 11, 12 是等价的, 它们应属于同一个等价类。不过, 这三个元素还不能构成一个等价类, 因为还有其他的元素与它们等价 (如 7)。所以  $\{1, 11, 12\}$  不是等价元素的最大集合。集合  $\{1, 2, 7, 8, 11, 12\}$  才是一个等价类。关系  $R$  还定义了另外两个等价类:  $\{3, 4, 5, 9, 13, 14\}$  和  $\{6, 10\}$ 。

在离线等价类 (offline equivalence class) 问题中, 已知  $n$  和  $R$ , 确定所有的等价类。注意每个元素只能属于某一个等价类。在在线等价类 (online equivalence class) 问题中, 初始时有  $n$  个元素, 每个元素都属于一个独立的等价类。需要执行以下的操作: 1)  $\text{Combine}(a, b)$  把包含  $a$  和  $b$  的等价类合并成一个等价类。2)  $\text{Find}(e)$  确定哪个类包含元素  $e$ , 搜索的目的是为了确定给定的两个元素是否在同一个类之中。因此, 对于同一类中的元素,  $\text{Find}$  将返回相同的结果, 而对不同类的元素, 则返回不同的结果。

可以利用两个  $\text{Find}$  操作和一个  $\text{Union}$  操作产生一个组合操作, 该操作能把两个不同的类合并成一个类。因此  $\text{Combine}(a,b)$  等价于:

```
i = Find(a); j = Find(b);
if (i != j) Union(i, j);
```

注意, 利用  $\text{Find}$  和  $\text{Union}$  操作, 可以向  $R$  中添加新关系。例如, 为了添加关系  $(a, b)$ , 可以首先判断  $a$  和  $b$  是否已经位于同一个等价类, 如果是, 则新关系是冗余的, 如果不是, 则对包含  $a$  和  $b$  的两个类执行  $\text{Union}$  操作。

本节主要关心在线等价类问题, 这类问题通常又称之为 union-find 问题。本节给出的解决方案很简单, 但是效率不是最高的。8.10.2 节给出了一个更快的解决方案。“离线等价类”问题的快速解决方案将在 5.5.5 节给出。

例3-5 [根据最后期限进行调度] 某工厂有一台机器能够执行  $n$  个任务, 任务  $i$  的释放时间为  $r_i$  (是一个整数), 最后期限为  $d_i$  (也是整数)。在该机上完成每个任务都需要一个单元的时间。一种可行的调度方案是为每个任务分配相应的时间段, 使得任务  $i$  的时间段正好位于释放时间和最后期限之间。一个时间段不允许分配给多个任务。

考察下面的四个任务:

任务	1	2	3	4
释放时间	0	0	1	2
最后期限	4	4	2	3

任务1和任务2的释放时间为0，任务3的释放时间为1，任务4的释放时间为2。下面的任务-时间调度方案是可行的：在0~1期间执行任务1；1~2期间执行任务3；2~3期间执行任务4；3~4期间执行任务2。进行调度的一种直观方法如下：

1) 按释放时间的递増次序对任务进行排序。

2) 考察1) 中所得到的任务序列。对于每个任务，确定与最后期限最接近的空闲时间段（位于最后期限之前）。如果这个空闲时间段位于任务的释放时间之前，则失败，否则把这个时间段分配给任务。

练习83要求你证明，如果不存在一个可行的调度方案，则上述策略将失败。

在线等价类问题的方法可用来实现2)。令 $d$ 为所有任务中最后一个的完成期限，各可用时间段的形式为“从 $i-1$ 至 $i$ ”，其中 $1 \leq i \leq d$ ，这些时间段被称为“时间段1，时间段2，...，时间段 $d$ ”。对于任意时间段 $a$ ，定义 $near(a)$ 为最大 $i$ ： $i \leq a$ 且时间段 $i$ 空闲。如果不存在这样的 $i$ ，则定义 $near(a) = near(0) = 0$ 。当且仅当 $near(a) = near(b)$ 时，两个时间段 $a$ 和 $b$ 属于同一个等价类。

在调度任务之前，对于所有时间段有 $near(a) = a$ ，且每个时间段都是一个独立的等价类。当时间段 $a$ 被分配给2) 中的某个任务时，对于所有 $near(b) = a$ 的时间段 $b$ ，其 $near$ 值发生变化，对于这些时间段，其新的 $near$ 值为 $near(a-1)$ 。因此，当时间段 $a$ 被分配给一个任务时，需要对当前包含时间段 $a$ 和 $a-1$ 的等价类进行合并（执行Union操作）。如果每个等价类 $E$ 用 $N[E]$ 表示， $near$ 的值是类中的成员，那么 $near(a)$ 将由 $N[Find(a)]$ 给出。（假定类名就是Find操作所返回的内容）。

例3-6 [布线] 一个电路由构件、针脚和电线构成。图3-18给出了一个由三个构件A，B和C组成的电路。每根电线连接了一对针脚。当且仅当要么有一根电线直接连接了 $a$ 和 $b$ ，要么存在一个针脚序列 $a_1, a_2, \dots, a_k$ ，使得 $a, a_1$ ； $a_1, a_2$ ； $a_2, a_3$ ；...； $a_{k-1}, a_k$ ；和 $a_k, b$ 均由电线直接相连时，两个针脚 $a$ 和 $b$ 是电子等价（electrically equivalent）的。网组(net)是指电子等价针脚的最大集合，“最大”是指不存在网组外的针脚与网组内的针脚电子等价。

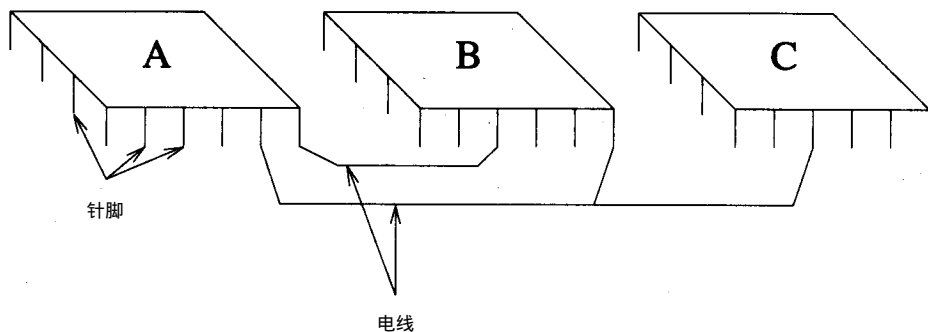


图3-18 一个印刷电路板上的3芯片电路

考察图3-19所示的电路。在该图中仅绘出了针脚和电线。14个针脚从1至14编号。每根电线可由它所连接的两个针脚来描述。例如，连接针脚1和11的电线可以表示为(1,11)，它与(11,1)等价。电线的集合为{(1,11)，(7,11)，(2,12)，(12,8)，(11,12)，(3,13)，(4,13)，(13,14)，(14,9)，(5,14)，(6,10)}。因此，该电路中所存在的网组如下：{1, 2, 3, 8, 11, 12}，{3, 4, 5, 9, 13, 14}和{6, 10}。

在离线网组搜索（offline net finding）问题中，已知针脚和电线，需要确定相应的网组。如果把每个针脚看成 $U$ 的一个成员，把每根电线看成 $R$ 的成员，那么离线网组搜索问题与离线

等价类问题完全相同。

对于在线网组搜索 (online net finding) 问题, 起始时有一组针脚的集合, 没有电线, 然后执行以下操作: 1) 增加一根连接  $a$  和  $b$  的电线; 2) 搜索包含针脚  $a$  的网组。搜索的目的是为了确定两个针脚是否位于同一个网组中。在线网组搜索问题实际上等同于在线等价类问题。初始时没有电线, 相当于  $R=\phi$ , 网组搜索操作对应于等价类的 Find 操作, 添加电线  $(a, b)$  对应于 Combine( $a, b$ )。

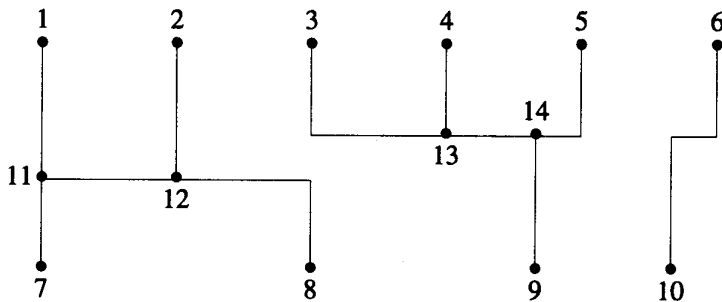


图3-19 仅给出针脚和电线的电路

## 2. 第一种解决方案

在线等价类问题的一种简单解决办法是使用一个数组  $E$  并令  $E(e)$  代表包含元素  $e$  的等价类。完成初始化、合并及搜索操作的函数如程序 3-46 所示。  $N$  是元素的数目,  $n$  和  $E$  均被定义为全局变量。为了合并两个不同的类, 可从类中任取一个元素, 然后把该类中所有元素的  $E$  值修改成另一个类中元素的  $E$  值。Initialize 和 Union 函数的复杂性均为  $\Theta(n)$  (假定在 Initialize 中调用 new 时不产生异常), Find 的复杂性为  $\Theta(1)$ 。从例 3-5 和例 3-6 中可以看出, 在应用这些函数时, 通常执行一次初始化,  $u$  次合并和  $f$  次搜索, 故所需要的总时间为  $\Theta(n+u*n+f)=\Theta(u*n+f)$ 。

程序 3-46 使用数组的在线等价类函数

```
void Initialize(int n)
{
    // 初始化  $n$  个类, 每个类仅有一个元素
    E = new int [n + 1];
    for (int e = 1; e <= n; e++)
        E[e] = e;
}

void Union(int i, int j)
{
    // 合并类  $i$  和类  $j$ 
    for (int k = 1; k <= n; k++)
        if (E[k] == j) E[k] = i;
}

int Find(int e)
{
    // 搜索包含元素  $i$  的类
    return E[e];
}
```

## 3. 第二种解决方案

针对每个等价类设立一个相应的链表, 可以降低合并操作的时间复杂性, 因为可以沿着类

j的链表找到所有 $E[k]=j$ 的元素，而不必去检查所有的E值。事实上，如果知道每个等价类的大小，可以改变较小类的E值，这样可以使合并操作更快。通过使用模拟指针，可以快速地访问代表元素e的节点。使用以下的约定：

- EquivNode是一个类，E，size和link是其私有数据成员，这些数据成员的类型均为int。
- 函数Initialize, Union和Find均为EquivNode的友元。
- node[1:n]用于描述n个元素（每个元素都有一个对应的等价类链表）。
- node[e].E既是Find(e)返回的值，也是一个指针，该指针指向类 node[e].E对应链表的首节点。

• 仅当e是链表的首节点时，才定义node[e].size，这时，它的值表示从node[e]开始，链表中的节点数目。

• node[e].link给出了包含节点e的链表的下一个节点。由于所使用的节点被编号为1至n，故可以用0而不是-1来表示空指针。

程序3-47给出了Initialize和Union的新代码。Find的代码与程序3-46相同。

程序3-47 使用链表的在线等价类函数

```
void Initialize(int n)
{ // 初始化n个类，每个类仅有一个元素
  node = new EquivNode [n + 1];
  for (int e = 1; e <= n; e++) {
    node[e].E = e;
    node[e].link = 0;
    node[e].size = 1;
  }
}

void Union(int i, int j)
{ // 合并类 i 和类 j
  // 使 i 代表较小的类
  if (node[i].size > node[j].size)
    swap(i, j);
  // 改变较小类的 E 值
  int k;
  for (k = i; node[k].link; k = node[k].link)
    node[k].E = j;
  node[k].E = j; // 链尾节点
  // 在链表j的首节点之后插入链表 i
  // 并修改新链表的大小
  node[j].size += node[i].size;
  node[k].link = node[j].link;
  node[j].link = i;
}

int Find(int e)
{ // 搜索包含元素 i 的类
  return node[e].E;
}
```

在使用链表时，因为一个等价类的大小为  $O(n)$ ，因此合并操作的复杂性为  $O(n)$ ，而初始化和搜索操作的复杂性仍分别保持为  $\Theta(n)$  和  $\Theta(1)$ 。为了确定1次初始化操作、 $u$  次合并操作和  $f$



次搜索操作所需要的时间复杂性，需要使用如下的定理。

定理3-1 如果开始时有 $n$ 个类，每个类有一个元素，则在执行 $u$ 次合并操作以后，

- 任何一个类的元素数都不会超过 $u+1$ 。
- 至少存在 $n-2u$ 个单元素类。
- $u < n$ 。

证明 见练习81。

1次初始化和 $f$ 次搜索的复杂性为 $\Theta(n+f)$ 。对于 $u$ 次合并，每次合并操作的开销为 $\Theta$ (较小类的大小)。令 $i$ 表示合并操作中的较小类。在合并期间， $i$ 中的每个元素从类 $i$ 移向类 $j$ ，因此 $u$ 次合并的复杂性由移动元素的总次数确定。移动类 $i$ 之后，新类的大小至少是类 $i$ 的两倍（因为在移动前有 $\text{size}[i] \leq \text{size}[j]$ ，而移动之后新类的大小为 $\text{size}[i]+\text{size}[j]$ ）。因此，由于在操作结束时没有哪个类的元素数会超过 $u+1$ （定理3-1a），所以在 $u$ 次合并期间，没有哪个元素的移动次数超过 $\log_2(u+1)$ 。另外，根据定理3-1b，最多可以移动 $2u$ 个元素，所以，元素移动的总次数不会超过 $2u \log_2(u+1)$ 。至此可以知道执行 $u$ 次合并操作所需要的时间为 $O(u \log u)$ 。1次初始化、 $u$ 次合并操作和 $f$ 次搜索的复杂性为 $O(n+u \log u+f)$ 。

### 3.8.4 凸包

多边形（polygon）是指至少有三条直线边的平面封闭图形。图3-20a所示的多边形有六条边，图3-20b中的多边形有八条边。多边形既包含其边线上的点，也包含由边所包围的所有点。凸多边形（convex polygon）的任意两个点（位于边线上或多边形内）之间的连线都包含在多边形内。图3-20a中的多边形是凸多边形，而图3-20b中的多边形为非凸多边形（nonconvex polygon）。图3-20b中画出了两条虚线段，虽然这两条虚线段的端点都在边线上或多边形内，但它们都包含了多边形以外的点。

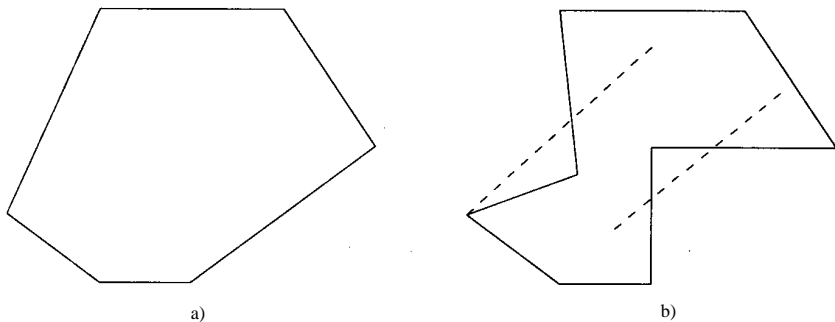


图3-20 凸多边形和非凸多边形

a) 凸多边形 b) 非凸多边形

平面上一个点集 $S$ 的凸包（convex hull）是指包含所有这些点的最小凸多边形，该多边形的角是 $S$ 的极点（extreme point）。图3-21给出了13个点（同一平面上），相应的凸包是由实线连成的多边形，所有的极点用圆圈来标记。如果 $S$ 中的所有点都落在一条直线上，则是一种退化情形，此时凸包被定义为包含所有点的最短直线。

寻找平面点集对应凸包的问题是计算几何中的基本问题。计算几何中另外几个问题（如寻找包含指定点集的最小矩形）的求解都需要借助于对凸包的计算。此外，凸包在图象处理和统



计学中都有很多应用。

假定在  $S$  对应的凸包内取一个点  $X$ ，然后从  $X$  向下画一条垂直线（如图3-22a所示）。练习85说明了如何选择点  $X$ 。令  $a_i$  表示这条垂直线与  $X$  和  $S$  中第  $i$  个点连线间的夹角（又称极角）。对  $a_i$  的测量是按逆时针方向进行的（即从垂线开始沿逆时针方向测量至  $X$  与第  $i$  个点的连线）。图3-22a 中给出了夹角  $a_2$ 。现在按照  $a_i$  的递增次序来排列  $S$  中的点，对于具有相同极角的点，再按照它们与  $X$  之间的距离来进行排列。在图3-22a 中，所有点按照上述次序被依次编号为1至13。

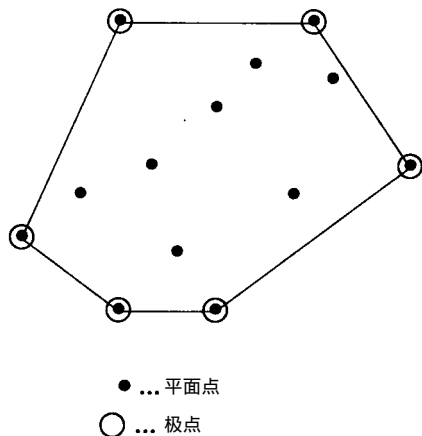


图3-21 平面点对应的凸包

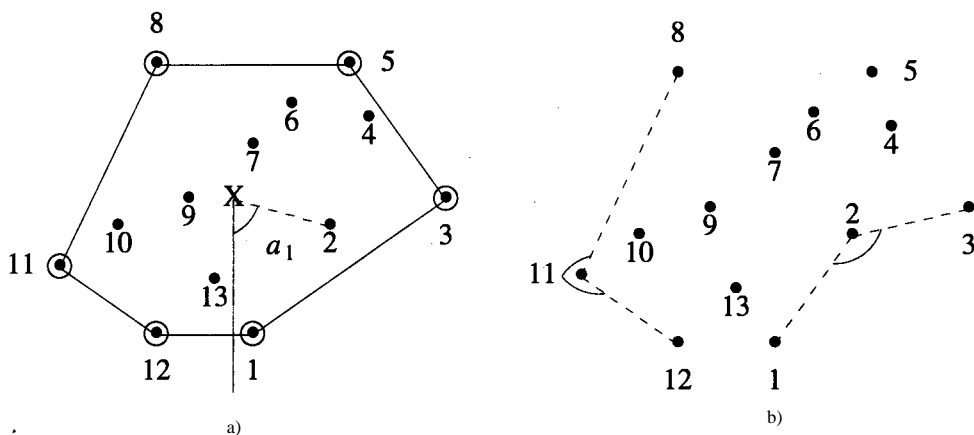


图3-22 识别极点

a) 顶点排序 b) 逆时针夹角

$X$  向下的垂线沿逆时针扫描会依次（按照极角的  $a_i$  次序）遇到  $S$  中的极点。如果  $u, v$  和  $w$  是按照逆时针排列的三个连续的极点，那么  $u$  到  $v$  与  $w$  到  $v$  两条连线之间的逆时针夹角将大于  $180$  度。（图3-22b 给出了点8, 11, 12之间的逆时针夹角）。当按照极角次序排列的3个连续点之间的逆时针夹角小于或等于  $180$  度时，那么其中的第二个点不是极点。注意当  $u, v, w$  间的夹角小于  $180$  度时，如果从  $u$  走到  $v$  再走到  $w$ ，那么在  $v$  点将会向右转。当按逆时针方向在一个凸多边形上行走时，所有的方向变化都是向左转。根据这种观察结果得到了图3-23的算法，该算法用于寻找极点以及  $S$  对应的凸包。

1) 用于处理退化的情形，即  $S$  中的点数为0或1，或者  $S$  中的所有点共线。1) 可以在  $\Theta(n)$  时间内完成，其中  $n$  是  $S$  的点。对于共线的判断，方法是，任取两个点，求出两点之间连线的方程式，然后检查余下的  $n-2$  个点，并判断它们是否在这条直线上。在这个过程中，如果所有点都是共线的，还可以同时确定包含这些点的最短直线的端点。

在2) 中，按照极角的次序排列所有的点，并把它们放入一个双向链表之中（之所以采用双

向链表是因为在3)中,需要消除不是极点的点),并在链表中反向移动,而这两种操作对于双向链表来说都是能直接实现的。练习85会要求你采用一个单向链表。因为需要排序,所以如果采用第2章中的排序算法,需要耗时 $\Theta(n^2)$ 。在第9章和第14章中,可以在 $O(n \log n)$ 时间内完成排序,因此2)的时间复杂性可以计为 $O(n \log n)$ 。

```

1) [处理退化情形]
   如果S的点数少于3个,则返回S
   如果所有点都在同一条直线上,则计算出包含所有点的最短直线的端点,并返回这两个端点

2) [按极角排序]
   在S对应的凸包内寻找点X
   按照极角对S中的点进行排序,若极角相同,则按与X的距离排序
   建立一个双向循环链表,把各个点按上述次序放入链表之中
   令right指向排序中的下一个点, left指向前一个点

3) [删除不是极点的点]
   令p为具有最小y坐标的点(也可以选择具有最大x坐标的点)
   for(x=p, rx=x的下一个点; x!=rx; ){
       rrx= rx的下一个点;
       if( x,rx,rrx间的夹角小于或等于180度){
           从链表中删除rx;
           rx=x; x=rx的前一个节点; }
       else {x=rx; rx=rrx;}
   }

```

图3-23 寻找与S对应的凸包的伪代码

在3)中,依次检查按逆时针次序排列的三个连续点,判断三个点构成的夹角是否小于或等于180度,如果是,则中间点rx不是极点,从链表中删除之。如果夹角超过180度,则第三点rrx可能是也可能不是极点,可从点x转向处理下一个点。当for循环终止时,双向循环链表中的每个点都满足属性:x,rx和rrx间的夹角超过180度。因此链表中的所有点都是极点。根据链表的right域遍历链表,可以得到逆时针排列的凸包的边界。从具有最小y坐标的点开始的,因为这个点肯定位于凸包之中。

对于3)的复杂性,注意到,在for循环中每次检查一个夹角之后会出现以下情形:1)顶点rx被删除,x在链表中后移一个位置,或2)x前移一个位置。由于被删除的顶点数为 $O(n)$ ,x最多向后移动 $O(n)$ 个位置。因此情形2)只会发生 $O(n)$ 次,因而for循环将执行 $O(n)$ 次。由于检查一个夹角需要耗时 $\Theta(1)$ ,所以3)的复杂性为 $O(n)$ 。这样,为了找到n个点的凸包,需要耗时 $O(n \log n)$ 。

## 练习

76. 程序3-43是稳定排序吗?

77. 比较程序3-43和3-48所给出的箱子排序函数的运行时间,使用 $n=10\ 000$ , $50\ 000$ 和 $100\ 000$ 进行测试。指出由于引入类Chain所产生的开销。

78. 给定一个n节点的链表,节点类型为ChainNode<Node>。按照score域对链表进行排序。

1) 编写一个函数,采用基数排序方法进行排序。函数的输入为:欲排序的链表、基数r、按基数r分解的数字位数d。函数的复杂性应为 $\Theta(d(r+n))$ 。试证明之。

2) 通过编译和执行(使用自己设计的测试数据)来测试函数的正确性。

3) 把所编写的函数与用链表完成插入排序的函数进行性能比较。可使用  $n=100, 1000, 10\ 000$ ;  $r=10$ 和 $d=3$ 来测量运行时间。

79. 1) 编写一个函数, 使用 $r=n$ 的基数排序算法对 $n$ 个 $0 \sim n^c-1$ 范围内的整数进行排序。函数的复杂性应为 $\Theta(cn)$ 。试证明之。假定函数的输入为待排序的链表。

2) 测试函数的正确性。

3) 对于 $n=10, 100, 1000, 10\ 000$ 和 $c=2$ , 测量函数的运行时间。用表格和图的形式给出测量结果。

80. 给定 $n$ 堆卡片, 每张卡片有三个域: 卡片所在堆的编号、卡片样式及卡片面值。每堆卡片最多有52张卡片, 因此卡片总数最多为 $52n$ 。可以假定每一堆至少有一张卡片, 所以卡片总数至少为 $n$ 。

1) 解释如何按照堆的编号对卡片进行排序(对于堆号相同的卡片按卡片样式进行排序, 对于样式也相同的卡片按其面值进行排序)。可以采用三次箱子排序过程来完成卡片排序。

2) 编写一段程序, 其输入为 $n$ 和卡片堆, 输出为排序后的卡片堆。把卡片堆表示成一个链表, 链表中的每个节点包含如下域: deck, suit, face和link。程序的复杂性应为 $\Theta(n)$ , 试证明之。

3) 测试程序的正确性。

81. 证明定理3-1。

82. 对于例3-6的在线网组搜索问题, 编写一个C++函数。要求模仿在线等价类问题进行处理, 并要求使用链表。测试程序的正确性。

83. 证明例3-5中所给出的策略: 仅当不存在可行的调度方案时, 搜索过程才会失败。

84. 对于例3-5的调度问题编写一个C++程序, 按照在线等价类问题进行处理, 要求使用链表。测试程序的正确性。

85. 1) 令 $u, v, w$ 是平面上的三个点, 假定这三点不在同一直线上。编写一个函数, 该函数从这三个点构成的三角形中取一个点。

2) 令 $S$ 是一个平面点集。编写一个函数来判断 $S$ 中的所有点是否共线。如果共线, 计算出包含所有这些点的最短直线的端点。如果不共线, 从点集中找出三个不共线的点。利用1)中的三个点以及函数来指定一个 $S$ 凸包内的点。函数的复杂性应为 $\Theta(n)$ , 试证明之。

3) 使用1)和2)中的代码把图3-23中的算法细化成一个C++程序, 程序的输入为点集 $S$ , 输出为点集 $S$ 对应的凸包。在输入 $S$ 期间, 可同时把点放入双向链表之中(之后将按照极角对这些点进行排序)。对于排序, 使用第2章所给出的排序算法, 或者使用复杂性为 $O(n \log n)$ 的排序算法。

4) 编写其他的凸包程序, 采用a) 单向链表, b) 基于公式的线性表来替代双向链表。

5) 测试程序的正确性。

86. 令 $c$ 是一个链表。假定在链表中向右移动时, 把链表指针的方向变反, 因此, 如果正处于节点 $p$ , 链表将被分成两个子链表。一个子链表将从 $p$ 节点开始, 一直延续到 $c$ 的最后一个节点。另一个子链表从链表 $c$ 中 $p$ 的前一个节点 $l$ 开始, 一直向后延续到 $c$ 的首节点。初始时,  $p=c, l=0$ 。

1) 绘出有6个节点的链表, 给出 $p$ 为第三个节点,  $l$ 为第二个节点的情形。

2) 编写一个函数使 $l$ 和 $p$ 前进一个节点。

3) 编写一个函数使 $l$ 和 $p$ 后退一个节点。

4) 用适当的数据来测试代码。

87. 使用单向链表来完成练习 85。用练习 86 的思想来确保图 3-23 中步骤 3 的 for 循环具有复杂性  $\Theta(n)$ 。

88. 采用练习 86 的思想来扩充程序 3-8 中 Chain 类的定义，使得能够在单向链表中高效地前进和后退。为此，按照练习 86 应增加私有成员 l 和 p，同时增加以下共享函数：

- 1) Reset——把 p 置为 first，l 置为 0。
- 2) Current(x)——返回由 p 所指向的元素 x；如果操作失败，则引发一个异常。
- 3) End——如果 p 指向链表尾部，则返回 true，否则返回 false。
- 4) Front——如果 p 指向链表首部，则返回 true，否则返回 false。
- 5) Next——使 p 和 l 右移一个位置；如果操作失败，则引发一个异常。
- 6) Previous——使 p 和 l 左移一个位置；如果操作失败，则引发一个异常。

对于扩充后的类定义，编写相应的 C++ 代码。为了高效地实现 Insert，Delete 和 Find 函数，可以引入另外一个私有成员 current，它给出由 p 所指向的元素的索引（即表中的第 1 个元素、第 2 个元素等）。用适当的数据来测试代码。

注：带 \* 号的练习题为提高题，较难。

\*89. 给出一种整数的表示，它适合于对任意大的整数进行算术运算，而且算术运算的结果没有精度损失。编写一个 C++ 程序来输入和输出大的整数，同时实现以下算术操作：加、减、乘和除。除法函数应返回两个整数：商和余数。此外，应编写一个函数用以释放一个整数（即释放该整数所占用的空间）。

\*90. 一个阶数为 d 的一元多项式（univariate polynomial）形式如下：

$$c_d x^d + c_{d-1} x^{d-1} + c_{d-2} x^{d-2} \dots + c_0$$

其中  $c_0 \neq 0$ ， $c_i$  是系数， $e_i$  是指数。根据定义，指数是非负整数。每个  $c_i x^i$  是多项式的一个项。我们希望设计一个模板类来支持涉及多项式的算术运算。为此，需要把每个多项式表示成一个由系数构成的线性表  $(c_0, c_1, c_2, \dots, c_d)$ 。

设计一个 C++ 模板类 Polynomial<T>，其中 T 给出了系数的类型。类 Polynomial 应该带有一个私有成员 degree，它是多项式的阶数。当然，它还可能包含其他的私有成员。多项式类应支持以下操作：

- 1) Polynomial()——创建一个 0 阶多项式。这个多项式的阶数为 0，不包含任何项。它是类的构造函数。
- 2) Degree()——返回多项式的阶数。
- 3) Input()——读入一个多项式。可以假定输入是由多项式的阶数和一个系数表构成，系数表中的系数按指数递增的次序排列。
- 4) Output()——输出多项式。输出格式可以与输入格式相同。
- 5) Add(b)——把当前多项式加到多项式 b 上，并返回所得结果。
- 6) Subtract(b)——减去多项式 b 并返回所得结果。
- 7) Multiply(b)——乘以多项式 b 并返回所得结果。
- 8) Divide(b)——除以多项式 b 并返回所得结果。
- 9) Value(x)——返回按 x 计算出的多项式的值。

对于 3) 至 9)，需要重载操作符 <<、>>、+、-、\*、/ 和 ()。对于 9)，语法 P(x) 应返回多项式在 x 点的取值，其中 P 的类型为 Polynomial。对程序进行测试。

\*91. 设计一个链表类来表示和处理一元多项式（见练习 90）。使用带头节点的循环链表。

每个节点的data域包含一个系数域和一个指数域。除了头节点外，多项式的循环链表的，每个节点均对应于多项式的一个项（系数不为0），系数为0的项不需考虑。所有的项按照指数的递减次序排列。头节点的指数域取值为-1。图3-24给出了一些示例。

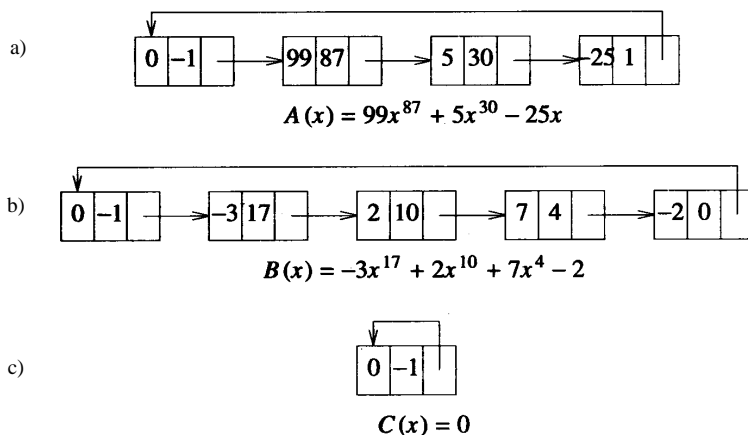


图3-24 多项式举例

一个一元多项式的外部表示（即针对输入和输出）将被假定为形如  $n, e_1, c_1, e_2, c_2, e_3, c_3, \dots, e_n, c_n$  的序列，其中  $e_i$  表示指数， $c_i$  表示系数， $n$  给出了多项式的项数。指数按照递减的次序排列，即有： $e_1 > e_2 > \dots > e_n$ 。

类应能支持练习90中的所有函数。采用适当的多项式测试代码。

### 3.9 参考及推荐读物

关于C++数据结构的其他参考书如下：

- 1) E.Horowitz, S.Sahni, D.Mehta. *Fundamentals of Data Structure in C++*. W.H.Freeman, 1994.
- 2) F.Carrano. *Data Abstraction and Problem Solving with C++*. Benjamin/Cummings, 1995.
- 3) A. Drozdek. *Data Structures and Algorithms in C++*. PWS, 1996.
- 4) M. Weiss. *Algorithms, Data Structures, and Problem Solving with C++*. Addison-Wesley, 1996.
- 5) T. Budd. *Classic Data Structures in C++*. Addison-Wesley, 1994.

China-pub.com

下载

## 第4章 数组和矩阵

在实际应用过程中，数据通常以表的形式出现。尽管用数组来描述表数据是最自然的方式，但有时为了减少程序的时间和空间需求，通常会采用自定义的描述方式，比如，当表中大部分数据全为0时。

本章首先检查了多维数组的行主描述形式和列主描述形式。通过行主描述和列主描述，可以把多维数组映射成一维数组。

尽管C++支持多维数组，但它无法保证数组下标的合法性。同时，C++也未能提供数组的输入、输出以及简单的算术运算（如数组赋值和数组相加）。为了克服这些不足，我们针对一维数组和二维数组分别设计了类Array1D和类Array2D。

矩阵通常被描述成一个二维数组。不过，矩阵的索引通常从1开始，而不是像数组那样从0开始，并且通常使用  $A(i,j)$  而不是  $A[i][j]$  来引用矩阵中的元素  $(i,j)$ 。为此，设计了另一个类Matrix 以便更好地描述矩阵。

本章还考察了一些具有特殊结构的矩阵，如对角矩阵、三对角矩阵、三角矩阵和对称矩阵。与采用二维数组描述矩阵相比，采用公式化的方法来描述这些特殊矩阵所需要的空间将大大减小，同时，公式化的描述方法还可以显著地节省诸如矩阵加和矩阵减操作所需要的运行时间。

本章的最后一节给出了稀疏矩阵（即大部分元素为0的矩阵）的公式化描述和链表描述，这两种描述方法对于0元素都做了特殊处理。

### 4.1 数组

#### 4.1.1 抽象数据类型

数据对象array的每个实例都是形如（index，value）的数据对集合，其中任意两对数据的index值都各不相同。有关数组的操作如下：

- *Create*——创建一个初始为空的数组（即：不含任何数据）
- *Store*——向数组中添加一对（index，value）数据，如果数组中已经存在索引值与index相同的数据对，则删除该数据对。
- *Retrieve*——返回具有给定index值的数据对的value值。

ADT4-1给出了具有上述三种操作的抽象数据类型Array。

ADT 4-1 数组的抽象数据类型描述

---

抽象数据类型Array{

实例

形如(index,value)的数据对集合，其中任意两对数据的index值都各不相同

操作

*Create()*：创建一个空的数组

*Store(index, value)*：添加数据(index, value)，同时删除具有相同index值的数据对（如果存在）

*Retrieve(index)*：返回索引值为index的数据对

---



例4-1 上个星期每天的高温（华氏度数）可用如下的数组来表示：

```
high={ (sunday, 82), (monday, 79), (tuesday, 85), (wednesday, 92), (thursday, 88), (friday, 89),
        (saturday, 91) }
```

数组中的每对数据都包含一个索引（星期）和一个值（当天的温度），数组的名称为 *high*。通过执行如下操作，可以将 *monday* 的温度改变为 83。

```
Store(monday, 83)
```

通过执行如下操作，还可以确定 *friday* 的温度：

```
Retrieve(friday)
```

也可以采用如下的数组来描述每天的温度：

```
high={ (0,82), (1,79), (2,85), (3,92), (4,88), (5,89), (6,91) }
```

在这个数组中，索引值是一个数值，而不是日期名，数值（0, 1, 2, ...）代替了一周中每天的名称（*sunday*, *monday*, *tuesday*, ...）。

#### 4.1.2 C++数组

尽管在 C++ 中数组是一个标准的数据结构，但 C++ 数组的索引（也称为下标）必须采用如下形式：

$$[i_1][i_2][i_3]\dots[i_k]$$

$i_j$  为非负整数。如果  $k$  为 1，则数组为一维数组，如果  $k$  为 2，则为二维数组。 $i_1$  是索引的第一个坐标， $i_2$  是第二个， $i_k$  是第  $k$  个。在 C++ 中，值为整数类型的  $k$  维数组 *score* 可用如下语句来创建：

```
int score[u_1][u_2][u_3]\dots[u_k]
```

其中  $u_j$  是正的常量或由常量表示的正的表达式。对于这样一个数组描述，索引  $i_j$  的取值范围为： $0 \leq i_j < u_j, 1 \leq j \leq k$ 。因此，该数组最多可以容纳  $n = u_1 u_2 u_3 \dots u_k$  个值。由于数组 *score* 中的每个值都是整数，所以需要占用 `sizeof(int)` 个字节，因而，整个数组所需要的内存空间为 `sizeof(score) = n * sizeof(int)` 个字节。C++ 编译器将为数组预留这么多空间。假如预留空间的起始地址为 *start*，则该空间将延伸至 `start + size(score) - 1`。

#### 4.1.3 行主映射和列主映射

为了实现与数组相关的函数 *Store* 和 *Retrieve*，必须确定索引值在 `[start, start + n * sizeof(score) - 1]` 中的相应位置。实际上就是把数组索引  $[i_1][i_2][i_3]\dots[i_k]$  映射到 `[0, n - 1]` 中的某个数  $map(i_1, i_2, i_3, \dots, i_k)$ ，使得该索引所对应的元素值存储在以下位置：

$$start + map(i_1, i_2, i_3, \dots, i_k) * \text{sizeof}(\text{int})$$

当数组维数为 1 时（即  $k = 1$ ），使用以下函数：

$$map(i_1) = i_1$$

当数组维数为 2 时，各索引可按图 4-1 所示的表格形式进行排列，第一个坐标相同的索引位于同一行，第二个坐标相同的索引位于同一列。

在图 4-1 中从第一行开始，依次对每一行中的每个索引从左至右进行连续编号，即可得到

图4-2a 所示的映射结果。这种把二维数组中的位置映射为  $[0, n-1]$  中某个数的方式被称为行主映射。C++中即采用了这种行主映射模式。图4-2b 中给出了另一种模式，称之为列主映射。在列主映射模式中，对索引的编号从最左列开始，每一列按照从上到下的次序进行排列。

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[0][5]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	[1][5]
[2][0]	[2][1]	[2][2]	[2][3]	[2][4]	[2][5]

图4-1 int score[3][6]的索引排列表

0	1	2	3	4	5		0	3	6	9	12	15
6	7	8	9	10	11		1	4	7	10	13	16
12	13	14	15	16	17		2	5	8	11	14	17
a)							b)					

图4-2 二维数组的映射

a) 行主映射 b) 列主映射

行主次序所对应的映射函数为：

$$\text{map}(i_1, i_2) = i_1 u_2 + i_2$$

其中  $u_2$  是数组的列数。可以注意到，在行主映射模式中，在对索引  $[i_1][i_2]$  进行编号时，第  $0, \dots, i_1-1$  行中的  $i_1 u_2$  个元素以及第  $i_1$  行中的前  $i_2$  个元素都已经被编号。

让我们用图4-2a 中的  $3 \times 6$  数组来验证上述的行主映射函数。由于列数为 6，所以映射公式变成：

$$\text{map}(i_1, i_2) = 6i_1 + i_2$$

因此有  $\text{map}(1,3)=6+3=9$ ， $\text{map}(2,5)=6*2+5=17$ 。与图4-2a 中所给出的编号相同。

可以扩充上述的行主映射模式来得到二维以上数组的映射函数。注意在行主次序中，首先列出所有第一个坐标为 0 的索引，然后是第一个坐标为 1 的索引，…。第一个坐标相同的所有索引按其第二个坐标的递增次序排列，也即各个索引按照词典序进行排列。对于一个三维数组，首先列出所有第一个坐标为 0 的索引，然后是第一个坐标为 1 的索引，…。第一个坐标相同的所有索引按其第二个坐标的递增次序排列，前两个坐标相同的所有索引按其第三个坐标的递增次序排列。例如数组 `score[3][2][4]` 中的索引按行主次序排列为：

[0][0][0], [0][0][1], [0][0][2], [0][0][3], [0][0][4], [0][1][0], [0][1][1], [0][1][2], [0][1][3],  
[1][0][0], [1][0][1], [1][0][2], [1][0][3], [1][0][4], [1][1][0], [1][1][1], [1][1][2], [1][1][3],  
[2][0][0], [2][0][1], [2][0][2], [2][0][3], [2][0][4], [2][1][0], [2][1][1], [2][1][2], [2][1][3]

三维数组的行主映射函数为：

$$\text{map}(i_1, i_2, i_3) = i_1 u_2 u_3 + i_2 u_3 + i_3$$

可以观察到，所有第一个坐标为  $i_1$  的元素都位于第一个坐标小于  $i_1$  的元素之前，第一个坐标都相同的元素数目为  $u_2 u_3$ 。因此第一个坐标小于  $i_1$  的元素数目为  $i_1 u_2 u_3$ ，第一个坐标等于  $i_1$  且第二个坐标小于  $i_2$  的元素数目为  $i_2 u_3$ ，第一个坐标等于  $i_1$  且第二个坐标等于  $i_2$  且第三个坐标小于  $i_3$  的元素数目为  $i_3$ 。

#### 4.1.4 类Array1D

尽管C++支持一维数组，但这种支持很不够。例如，可以使用超出正常范围之外的索引值来访问数组。考察如下的数组a：

```
int a[9]
```

可以访问数组元素a[-3]，a[9]和a[90]，尽管-3，9和90是非法的索引。允许使用非法的索引通常会使程序产生无法预料的行为并给调试带来困难。并且C++数组不能使用如下的语句来输出数组：

```
cout << a << endl;
```

也不能对一维数组进行诸如加法和减法等操作。为了克服这些不足，定义了类Array1D(见程序4-1)，该类的每个实例X都是一个一维数组。X的元素存储在数组X.element之中，第i个元素位于X.element[i]，0 ≤ i < size。

程序4-1 一维数组的类定义

```
template<class T>
class Array1D {
public:
    Array1D(int size = 0);
    Array1D(const Array1D<T>& v); // 复制构造函数
    ~Array1D() {delete [] element;}
    T& operator[](int i) const;
    int Size() {return size;}
    Array1D<T>& operator=(const Array1D<T>& v);
    Array1D<T> operator+() const; // 一元加法操作符
    Array1D<T> operator+(const Array1D<T>& v) const;
    Array1D<T> operator-() const; // 一元减法操作符
    Array1D<T> operator-(const Array1D<T>& v) const;
    Array1D<T> operator*(const Array1D<T>& v) const;
    Array1D<T>& operator+=(const T& x);
private:
    int size;
    T *element; // 一维数组
};
```

这个类的共享成员包括：构造函数，复制构造函数，析构函数，下标操作符[]，返回数组大小的函数Size，算术操作符+、-、\*和+ =。此外还可以添加其他的操作符。程序4-2给出了构造函数和复制构造函数的代码。构造函数有点违背ANSI C++的要求，它允许数组具有0个元素。如果我们不希望出现这种违规行为，可以对这段代码进行适当的修改。

程序4-2 一维数组的构造函数

```
template<class T>
Array1D<T>::Array1D(int sz)
{ // 一维数组的构造函数
    if (sz < 0) throw BadInitializers();
    size = sz;
```

```
    element = new T[sz];
}
template<class T>
Array1D<T>::Array1D(const Array1D<T>& v)
{ // 一维数组的复制构造函数
    size = v.size;
    element = new T[size]; // 申请空间
    for (int i = 0; i < size; i++) // 复制元素
        element[i] = v.element[i];
}
```

程序4-3给出了重载操作符[]的代码。该操作符用来返回指向第i个元素的引用，这样可以使存储和查询操作按照很自然的方式进行。利用这个操作符，使用语句：

```
X[1] = 2 * Y[3];
```

就可以按照期望的方式进行工作，其中X和Y的类型均为Array1D。代码Y[3]在对象Y上施加操作符[]，结果返回指向元素3的一个引用，然后将它乘以2。代码X[1]也调用了操作符[]，结果返回一个指向X[1]的引用，之后2\*Y[3]的结果将存储在这个引用位置内。

程序4-3 重载下标操作符[]

```
template<class T>
T& Array1D<T>::operator[](int i) const
{ // 返回指向第 i 个元素的引用
    if (i < 0 || i >= size) throw OutOfBounds();
    return element[i];
}
```

程序4-4给出了赋值操作符的代码。这段代码通过验证等号左右两边是否相同来避免进行自我赋值（即X=X）。为了进行赋值，应该首先释放目标数组\*this所占用的空间，然后分配能够容纳源数组v的空间。尽管对new的调用可能会失败，但并没有捕获所引发的异常，而是把它留给更适合处理这种异常的代码来捕获。如果new没有引发异常，则把源数组中的元素逐个复制到目标数组中。

程序4-4 重载赋值操作符=

```
template<class T>
Array1D<T>& Array1D<T>::operator=(const Array1D<T>& v)
{ // 重载赋值操作符=
    if (this != &v) { // 不是自我赋值
        size = v.size;
        delete [] element; // 释放原空间
        element = new T[size]; // 申请空间
        for (int i = 0; i < size; i++) // 复制元素
            element[i] = v.element[i];
    }
    return *this;
}
```

程序4-5给出了二元减法操作符、一元减法操作符和增值操作符的代码。其他操作符的代

码与此类似。

程序4-5 重载二元减法操作符、一元减法操作符和增值操作符

```
template<class T>
Array1D<T> Array1D<T>:: operator-(const Array1D<T>& v) const
{// 返回 w = (*this) - v
    if (size != v.size) throw SizeMismatch();
    // 创建结果数组 w
    Array1D<T> w(size);
    for (int i = 0; i < size; i++)
        w.element[i] = element[i] - v.element[i];
    return w;
}

template<class T>
Array1D<T> Array1D<T>::operator-() const
{// 返回w = -(*this)
    // 创建结果数组 w
    Array1D<T> w(size);
    for (int i = 0; i < size; i++)
        w.element[i] = -element[i];
    return w;
}

template<class T>
Array1D<T>& Array1D<T>::operator+=(const T& x)
{//把 x 加到 (*this)的每个元素上
    for (int i = 0; i < size; i++)
        element[i] += x;
    return *this;
}
```

### 复杂性

当T是一个内部C++数据类型（如int, float和char）时，构造函数和析构函数的复杂性为 $\Theta(1)$ ，而当T是一个用户自定义的类时，构造函数和析构函数的复杂性为 $O(\text{size})$ 。之所以存在这种差别，是因为当T是一个用户自定义类时，在用new（delete）创建（删除）数组element的过程中，对于element的每个元素都要调用一次T的构造函数（析构函数）。下标操作符[ ]的复杂性为 $\Theta(1)$ ，其他操作符的复杂性均为 $O(\text{size})$ 。（注意复杂性不会是 $\Theta(\text{size})$ ，因为所有操作符的代码都可以引发一个异常并提前终止）。

#### 4.1.5 类Array2D

对于二维数组，可以定义一个类Array2D，见程序4-6。Array2D所采用的描述格式类似于图1-1。二维数组可被视为一维数组的集合。虽然在图1-1中采用一个一维数组指向各个行数组，但在程序4-6中，采用一维数组row来直接存储每个行数组。数组row之所以被设置成标准的C++一维数组而不是Array1D的一个实例，是因为我们能够完全控制对row中每个元素的访问（row是一个私有成员）。因此，可以确保使用合法的数组下标。我们不需要Array1D所提供的

附加功能。row[i]是类型为Array1D的一维数组，它代表二维数组的第i行。请注意，与图1-1不同的是，row[i]不是指向一维数组的指针。另外一种采用行主映射方式描述二维数组的方法见4.2.2节。

程序4-6 二维数组的类定义

```
template<class T>
class Array2D {
public:
    Array2D(int r = 0, int c = 0);
    Array2D(const Array2D<T>& m); // 复制构造函数
    ~Array2D() {delete [] row;}
    int Rows() const {return rows;}
    int Columns() const {return cols;}
    Array1D<T>& operator[](int i) const;
    Array2D<T>& operator=(const Array2D<T>& m);
    Array2D<T> operator+() const; // 一元加法操作符
    Array2D<T> operator+(const Array2D<T>& m) const;
    Array2D<T> operator-() const; // 一元减法操作符
    Array2D<T> operator-(const Array2D<T>& m) const;
    Array2D<T> operator*(const Array2D<T>& m) const;
    Array2D<T>& operator+=(const T& x);
private:
    int rows, cols; // 数组维数
    Array1D<T> *row; // 一维数组的数组
};
```

程序4-7给出了构造函数的代码。缺省情况下创建一个0行0列的数组，0行、非0列的数组是不允许的。如下语句：

```
row=new Array1D<T>[r];
```

创建一个具有r个位置的一维数组row，每个位置的类型为Array1D<T>。在创建数组row时，对于每个位置都将调用Array1D的构造函数。row[i]是一个具有缺省大小（为0）的一维数组，0 ≤ i < r。由于在创建一个数组时，仅会调用缺省的构造函数，因此在创建数组row的过程中存在一个for循环，在这个循环中将依次调整row中每个元素的大小。Resize是Array1D的一个新的成员函数，通过执行以下代码，它能把一个一维数组的大小变成sz：

```
delete [] element;
size = sz;
element = new T [size];
```

程序4-7 二维数组的构造函数

```
template<class T>
Array2D<T>::Array2D(int r, int c)
{ // 二维数组的构造函数
    // 合法的 r 和 c
    if (r < 0 || c < 0) throw BadInitializers();
    if (!(r || c) && (r || c))
```

```

        throw BadInitializers();
    rows = r;
    cols = c;
    // 分配 r个具有缺省大小的一维数组
    row = new Array1D<T> [r];
    // 调整每个元素的大小
    for (int i = 0; i < r; i++)
        row[i].ReSize(c);
}

```

注意，Array2D的析构函数并未明确释放分配给二维数组每一行元素的空间。不过，在执行`delete [ ] row`时，对于数组`row`的每一个元素，`delete`都将调用Array1D的析构函数，由Array1D的析构函数来释放每行元素所占用的空间。

程序4-8中的复制构造函数首先创建一个具有给定位置数的数组`row`，然后利用一维数组的赋值操作符复制二维数组中的每一行元素。与程序4-7中的构造函数不同的是，复制构造函数并未验证`m.rows`和`m.cols`的合法性，因为当`m`被创建时，这两个值都是合法的。赋值操作符的代码类似于复制构造函数的代码，不过，与程序4-4中的赋值操作符代码一样，Array2D的赋值操作也检查了是否为自我赋值。

程序4-8 二维数组的复制构造函数

```

template<class T>
Array2D<T>::Array2D(const Array2D<T>& m)
{ // 二维数组的复制构造函数
    rows = m.rows;
    cols = m.cols;
    // 分配指向一维数组的数组
    row = new Array1D<T> [rows];
    // 复制每一行
    for (int i = 0; i < rows; i++)
        row[i] = m.row[i];
}

```

若`X`的类型为`Array2D<T>`，则`X[i][j]`被分解为`(X.operator[i]).operator[j]`。因此，对于`X[i][j]`将首先利用实参`i`来调用`Array2D<T>::operator[ ]`，如果该操作返回了对象`Y`，则继续用实参`j`来调用`typeof(Y)::operator[ ]`。若`typeof(Y)`不同于`Array2D<T>`，则仅对二维数组的第一个索引调用`Array2D<T>::operator[ ]`。根据这种理解，可以得到程序4-9中的代码，这段代码只是简单地返回一个与第一个索引相对应的一维数组的引用。此时，若执行代码`X[i][j]`，则`X[i]`将调用`Array2D<T>::operator[ ]`，该操作符返回一个指向`X.row[i]`（即`X`的第`i`行）的引用。由于这个引用的类型为`Array1D<T>`，所以接下来将调用`Array1D<T>::operator[ ]`并返回一个指向相应数组元素的引用。

程序4-9 二维数组的重载操作符`[ ]`

```

template<class T>
Array1D<T>& Array2D<T>::operator[](int i) const
{ // 二维数组的第一个索引
    if (i < 0 || i >= rows) throw OutOfBounds();
}

```



```
    return row[i];
}
```

二元减法操作符的代码见程序 4-10，它只是简单地对二维数组的每一行调用 `Array1D<T>::operator-`。加法操作符、一元减法操作符、增值操作符和输出操作符的代码与此类似。

程序4-10 二元减法操作符

```
template<class T>
Array2D<T> Array2D<T>:: operator-(const Array2D<T>& m) const
{// 返回 w = (*this) - m.
    if (rows != m.rows || cols != m.cols)
        throw SizeMismatch();
    // 创建存放结果的数组 w
    Array2D<T> w(rows,cols);
    for (int i = 0; i < rows; i++)
        w.row[i] = row[i] - m.row[i];
    return w;
}
```

乘法操作符的实现与矩阵乘（见程序 2-25）很相似，见程序 4-11。

程序4-11 乘法操作符

```
template<class T>
Array2D<T> Array2D<T>:: operator*(const Array2D<T>& m) const
{// 矩阵乘，返回 w = (*this) * m.
    if (cols != m.rows) throw SizeMismatch();
    // 创建存放结果的数组 w
    Array2D<T> w(rows, m.cols);
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < m.cols; j++) {
            T sum = (*this)[i][0] * m[0][j];
            for (int k = 1; k < cols; k++)
                sum += (*this)[i][k] * m[k][j];
            w[i][j] = sum;
        }
    return w;
}
```

### 复杂性

当T是一个C++内部数据类型时，构造函数和析构函数的复杂性均为  $O(\text{rows})$ ，而当T是一个用户自定义的类时，构造函数和析构函数的复杂性为  $O(\text{rows} * \text{cols})$ 。复制构造函数及operator的复杂性为  $O(\text{rows} * \text{cols})$ ，下标操作符的复杂性为  $\Theta(1)$ ，乘法操作符的复杂性  $O(\text{rows} * \text{cols} * \text{m.cols})$ 。

### 练习

1. 扩充Array1D类（见程序 4-1），重载操作符 `<<`（输入一个数组）、`+`（一元加法）、`*=`（将每个元素右乘一个类型为T的元素）、`/=`和`-=`。试测试代码。

2. 对于一维数组设计一个类  $\text{Array1D}<\text{T1}, \text{T2}>$ ，其中  $\text{T1}$  是数组索引的类型， $\text{T2}$  是数组元素的类型。 $\text{T1}$  可以是任何枚举类型。例如，如果  $\text{T1}$  为  $\text{bool}$  类型，那么合法的索引值为  $\text{true}$  和  $\text{false}$ 。该类中应包含  $\text{Array1D}$ （见程序 4-1）的所有共享成员。

3. 对于  $\text{Array2D}$  类完成练习 1。另外再增加一个操作符，用来对两个大小相同的数组进行乘法操作（两个数组中相对应的元素两两相乘）。

4. 编写与二维数组的创建和删除函数（见程序 1-13 和程序 1-14）相对应的模板函数  $\text{Make3DArray}$  和  $\text{Delete3DArray}$ 。可使用如下语句创建一个整数数组  $s$ ：

```
int ***x
```

可以使用语法  $x[i][j][k]$  来访问每个元素。试测试代码的正确性。

5. 模仿  $\text{Array2D}<\text{T}>$  设计一个三维数组的类  $\text{Array3D}<\text{T}>$ 。

6. 按行主次序列出  $\text{score}[2][3][2][2]$  的索引。

7. 给出四维数组的行主映射函数。

8. 给出五维数组的行主映射函数。

9. 给出  $k$  维数组的行主映射函数。

10. 按列主次序列出  $\text{score}[2][3][4]$  的索引。注意，此时首先列出第三个坐标为 0 的所有索引，然后列出第三个坐标为 1 的所有索引，…。第三个坐标相同的索引按其第二个坐标的次序进行排列，后两个坐标都相同的索引按其第一个坐标的次序排列。

11. 给出三维数组的列主映射函数（参考前面的练习）。

12. 按列主次序列出  $\text{score}[2][3][2][2]$  的索引。

13. 给出四维数组的列主映射函数。

14. 给出  $k$  维数组的列主映射函数。

15. 假定对于一个二维数组采用从最后一行开始，每一行从右至左的方式进行映射

1) 按这种次序列出  $\text{score}[3][5]$  的索引。

2) 给出  $\text{score}[u_1][u_2]$  的映射函数。

16. 假定对于一个二维数组采用从最后一列开始，每一列从顶至底的方式进行映射

1) 按这种次序列出  $\text{score}[3][5]$  的索引。

2) 给出  $\text{score}[u_1][u_2]$  的映射函数。

## 4.2 矩阵

### 4.2.1 定义和操作

一个  $m \times n$  的矩阵（matrix）是一个  $m$  行、 $n$  列的表（如图 4-3 所示），其中  $m$  和  $n$  是矩阵的维数。

	列1	列2	列3	列4
行1	7	2	0	9
行2	0	1	0	5
行3	6	4	2	0
行4	8	2	7	3
行5	1	4	9	6

图4-3 一个  $5 \times 4$  的矩阵

矩阵通常被用来组织数据。例如，为了登记世界上的资源，可以首先列出一个感兴趣的资源类型表，在这个表中可以包含矿产（金、银等）、动物（狮子、大象等）、人（物理学家、工程师等）等。对于每一种资源可以确定在每个国家中现存的数量。可以把这些数据列在一个二维的表中，其中每一列对应于一个国家，每一行对应于一种资源。这样就得到了一个  $n$  列（对应于  $n$  个国家）、 $m$  行（对应于  $m$  种资源）的资源矩阵。可以使用符号  $M(i, j)$  来引用矩阵  $M$  中第  $i$  行、第  $j$  列  $1 \leq i \leq m, 1 \leq j \leq n$  的元素。如果上述资源矩阵中，第  $i$  行代表猫，第  $j$  列代表美国，那么  $assert(i, j)$  就代表美国所拥有的猫的总数。

对于矩阵来说，最常见的操作就是矩阵转置、矩阵加、矩阵乘。一个  $m \times n$  矩阵的转置矩阵是一个  $n \times m$  的矩阵  $M^T$ ，它与  $M$  之间存在以下关系：

$$M^T(i, j) = M(j, i), 1 \leq i \leq n, 1 \leq j \leq m$$

仅当两个矩阵的维数相同时（即具有相同的行数和列数），才可以对两个矩阵求和。两个  $m \times n$  矩阵  $A$  和  $B$  相加所得到的  $m \times n$  矩阵  $C$  如下：

$$C(i, j) = A(i, j) + B(i, j), 1 \leq i \leq n, 1 \leq j \leq m$$

仅当一个  $m \times n$  矩阵  $A$  的列数与另一个  $q \times p$  矩阵  $B$  的行数相同时（即  $n = q$ ），才可以执行矩阵乘法  $A * B$ 。 $A * B$  所得到的  $m \times p$  矩阵  $C$  满足以下关系：

$$C(i, j) = \sum_{k=1}^n A(i, k) * B(k, j), 1 \leq i \leq m, 1 \leq j \leq p$$

例4-2 考察上面的资源矩阵。假定有两家公司分别给出了一个资源矩阵，且其中的数据没有重复。若两个  $m \times n$  资源矩阵分别为  $assert1$  和  $assert2$ ，要得到所需要的资源矩阵，只需把矩阵  $assert1$  和  $assert2$  相加即可。

接下来，假定有另外一个  $m \times s$  矩阵  $value$ ， $value(i, j)$  代表资源矩阵中第  $i$  行、第  $j$  列资源的单价。令  $CV(i, j)$  代表国家  $i$  所拥有的资源  $j$  的总价值，则  $CV$  是一个  $n \times s$  矩阵，满足如下公式：

$$CV = assert^T * value$$

按照二维数组来计算矩阵的转置以及矩阵加和乘的 C++ 函数见第 2 章（见程序 2-19，2-22，2-24 和 2-25）。

#### 4.2.2 类 Matrix

可用如下的二维整数数组来描述元素为整数的  $m \times n$  矩阵  $M$ ：

```
int x[m][n]; 或 Array2D<int> x[m][n];
```

其中  $M(i, j)$  对应于  $x[i-1][j-1]$ 。这种形式要求使用数组的索引  $[ ]$  来指定每个矩阵元素。这种变化降低了应用代码的可读性，同时也增加了出错的概率。这可以通过定义一个类 `Matrix` 来克服，在 `Matrix` 类中，使用  $()$  来指定每个元素，并且各行和各列的索引值都是从 1 开始的。

在程序 4-12 的类 `Matrix` 中采用一个一维数组 `element` 来存储  $rows \times cols$  矩阵中的  $rows * cols$  个元素。

程序 4-12 类 `Matrix`

```
template<class T>
class Matrix {
```

```

public:
    Matrix(int r = 0, int c = 0);
    Matrix(const Matrix<T>& m); //复制构造函数
    ~Matrix() {delete [] element;}
    int Rows() const {return rows;}
    int Columns() const {return cols;}
    T& operator()(int i, int j) const;
    Matrix<T>& operator=(const Matrix<T>& m);
    Matrix<T> operator+() const; // 一元加法
    Matrix<T> operator+(const Matrix<T>& m) const;
    Matrix<T> operator-() const; // 一元减法
    Matrix<T> operator-(const Matrix<T>& m) const;
    Matrix<T> operator*(const Matrix<T>& m) const;
    Matrix<T>& operator+=(const T& x);
private:
    int rows, cols; // 矩阵维数
    T *element;    // 元素数组
};

```

程序4-13给出了构造函数的代码。复制构造函数与赋值操作符的代码相类似，可参考Array1D中的相应代码。

程序4-13 类Matrix的构造函数

```

template<class T>
Matrix<T>::Matrix(int r, int c)
{// 类Matrix的构造函数
    // 验证 r和 c的合法性
    if (r < 0 || c < 0) throw BadInitializers();
    if ((!r || !c) && (r || c))
        throw BadInitializers();
    // 创建矩阵
    rows = r; cols = c;
    element = new T [r * c];
}

```

为了重载矩阵下标操作符 ( ), 使用了C++的函数操作符 ( ), 与数组的下标操作符 []不同的是, 该操作符可以带任意数量的参数。对于一个矩阵来说, 需要两个整数参数。程序 4-14中的代码返回一个指向矩阵元素 (i, j) 的引用。

程序4-14 类Matrix的下标操作符

```

template<class T>
T& Matrix<T>::operator()(int i, int j) const
{// 返回一个指向元素 (i,j)的引用
    if (i < 1 || i > rows || j < 1 || j > cols) throw OutOfBounds();
    return element[(i - 1) * cols + j - 1];
}

```

程序4-15给出了减法操作符的代码。与一维数组的减法操作 (见程序 4-5) 和二维数组的

减法操作（见程序4-10）相比，程序4-15中矩阵减法操作更接近程序4-5。矩阵加法操作符、一元减法操作符、增值操作符和输出操作符的代码都比较类似。

程序4-15 类Matrix的减法操作符

```
template<class T>
Matrix<T> Matrix<T>:: operator-(const Matrix<T>& m) const
{// 返回 (*this) - m.
    if (rows != m.rows || cols != m.cols)
        throw SizeMismatch();
    // 创建结果矩阵 w
    Matrix<T> w(rows, cols);
    for (int i = 0; i < rows * cols; i++)
        w.element[i] = element[i] - m.element[i];
    return w;
}
```

尽管可以参照二维数组的乘法操作符代码（见程序4-11）来实现矩阵乘法，但它的效率可能比较低。程序4-16中矩阵乘法代码内的循环结构类似于程序2-25和程序4-11，它们都有三个嵌套的for循环。最内层的循环把\*this的第i行与m的第j列相乘，得到元素(i,j)。进入最内层循环时，element[ct]是第i行的第一个元素，m.element[cm]是第j列的第一个元素。为了得到第i行的下一个元素，可将ct增加1，因为在行主次序中同一行的元素是连续存放的。为了得到第j列的下一个元素，可将cm增加m.cols，因为在行主次序中同一列的两个相邻元素在位置上相差m.cols。当最内层循环完成时，ct指向第i行的最后一个元素，cm指向第j列的最后一个元素。对于for j循环的下一循环，起始时必须将ct指向第i行的第一个元素，而将cm指向m的下一列的第一个元素。对ct的调整是在最内层循环完成后进行的。当for j循环完成时，需要将ct指向下一行的第一个元素，而将cm指向第一列的第一个元素。

程序4-16 类Matrix的乘法操作符

```
template<class T>
Matrix<T> Matrix<T>:: operator*(const Matrix<T>& m) const
{// 矩阵乘法，返回 w = (*this) * m.
    if (cols != m.rows) throw SizeMismatch();
    Matrix<T> w(rows, m.cols); // 结果矩阵
    // 为 *this, m和 w定义游标
    // 并设定初始位置为(1,1)
    int ct = 0, cm = 0, cw = 0;
    // 对所有的i和j计算w(i,j)
    for (int i = 1; i <= rows; i++) {
        // 计算出结果的第 i 行
        for (int j = 1; j <= m.cols; j++) {
            // 计算w(i,j)的第一项
            T sum = element[ct] * m.element[cm];
            // 累加其余项
            for (int k = 2; k <= cols; k++) {
                ct++; // 指向*this第i行的下一个元素
```

```

    cm += m.cols; // 指向m 的第j 列的下一个项
    sum += element[ct] * m.element[cm];
}
w.element[cw++] = sum; // 保存w(i,j)
// 重新调整至行首和下一列
ct -= cols - 1;
cm = j;
}
// 重新调整至下一行的行首和第一列
ct += cols;
cm = 0;
}
return w;
}

```

### 复杂性

当T是一个内部数据类型时，矩阵构造函数复杂性为  $O(1)$ ，而当T是一个用户自定义的类时，构造函数的复杂性为  $O(\text{rows} \times \text{cols})$ 。复制构造函数的复杂性为  $O(\text{rows} \times \text{cols})$ ，下标操作符的复杂性为  $O(1)$ ，乘法操作符的复杂性  $O(\text{rows} \times \text{cols} \times \text{m.cols})$ 。所有其他矩阵操作符的渐进复杂性分别与Array2D类中对应操作符的渐进复杂性相同。

### 练习

17. 扩充Matrix类（见程序4-12），重载操作符  $<<$ （输入一个矩阵）、 $+$ （一元加法）、 $*=$ （将每个元素右乘一个类型为T的元素）、 $/=$ 和 $-=$ 。测试所编写的代码。

18. 扩充Matrix类，增加一个共享成员Transpose，用来返回转置后的矩阵。

19. 通过测量减法和乘法操作所需要的时间来比较Array2D类和Matrix类的性能。阐述采用行主映射的优点。

## 4.3 特殊矩阵

### 4.3.1 定义和应用

方阵（square matrix）是指具有相同行数和列数的矩阵。一些常用的特殊方阵如下：

- 对角矩阵（diagonal） $M$ 是一个对角矩阵当且仅当 $i \neq j$ 时有 $M(i, j)=0$ 。如图4-4a所示。
- 三对角矩阵（tridiagonal） $M$ 是一个三对角矩阵当且仅当 $|i - j| > 1$ 时有 $M(i, j)=0$ 。如图4-4b所示。

2 0 0 0	2 1 0 0	2 0 0 0	2 1 3 0	2 4 6 0
0 1 0 0	3 1 3 0	5 1 0 0	0 1 3 8	4 1 9 5
0 0 4 0	0 5 2 7	0 3 1 0	0 0 1 6	6 9 4 7
0 0 0 6	0 0 9 0	4 2 7 0	0 0 0 0	0 5 7 0
a)	b)	c)	d)	e)

图4-4  $4 \times 4$ 矩阵

a) 对角矩阵 b) 三对角矩阵 c) 下三角矩阵 d) 上三角矩阵 e) 对称矩阵

• 下三角矩阵 (lower triangular)  $M$  是一个下三角矩阵当且仅当  $i < j$  时有  $M(i, j) = 0$ 。如图 4-4c 所示。

• 上三角矩阵 (upper triangular)  $M$  是一个上三角矩阵当且仅当  $i > j$  时有  $M(i, j) = 0$ 。如图 4-4d 所示。

• 对称矩阵 (symmetric)  $M$  是一个对称矩阵当且仅当对于所有的  $i$  和  $j$  有  $M(i, j) = M(j, i)$ 。如图 4-4e 所示。

例4-3 考察佛罗里达州的六个城市 Gainesville, Jacksonville, Miami, Orlando, Tallahassee 和 Tampa。将这六个城市从1至6进行编号。任意两个城市之间的距离可以用一个  $6 \times 6$  的矩阵来表示。矩阵的第  $i$  行和第  $i$  列代表第  $i$  个城市,  $distance(i, j)$  代表城市  $i$  和城市  $j$  之间的距离。图4-5给出了相应的矩阵。由于对于所有的  $i$  和  $j$  有  $distance(i, j) = distance(j, i)$ , 所以该矩阵是一个对称矩阵。

	GN	JX	MI	OD	TL	TM
GN	0	73	333	114	148	129
JX	73	0	348	140	163	194
MI	333	348	0	229	468	250
OD	114	140	229	0	251	84
TL	148	163	468	251	0	273
TM	129	194	250	84	273	0

GN = Gainesville	OD = Orlando
JX = Jacksonville	TL = Tallahassee
MI = Miami	TM = Tampa

距离单位：公里

图4-5 城市距离矩阵

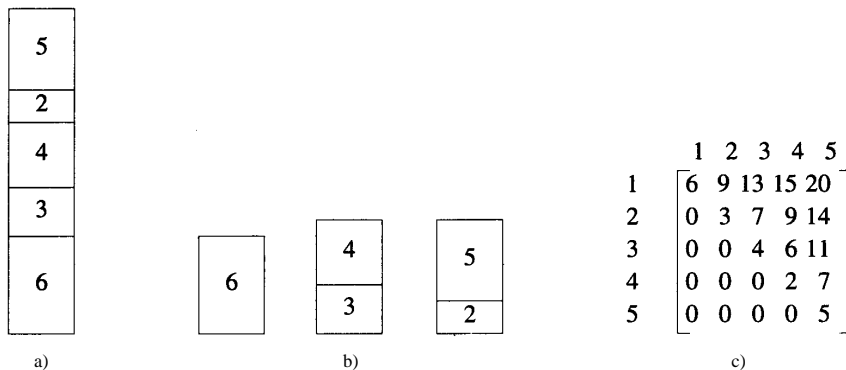


图4-6 堆栈折叠

a) 堆栈 b) 3个折叠堆栈 c) 矩阵

例4-4 假定有一个堆栈, 其中有  $n$  个纸盒, 纸盒1位于栈顶, 纸盒  $n$  位于栈底。每个纸盒的宽度为  $w$ , 深度为  $d$ 。第  $i$  个纸盒的高度为  $h_i$ 。堆栈的体积为  $w * d * \sum_{i=1}^n h_i$ 。在堆栈折叠 (stack folding) 问题中, 选择一个折叠点  $i$  把堆栈分解成两个子堆栈, 其中一个子堆栈包含纸盒 1至  $i$ , 另一个子堆栈包含纸盒  $i+1$ 至  $n$ 。重复这种折叠过程, 可以得到若干个堆栈。如果创建了  $s$  个堆栈, 则



这些堆栈所需要的空间宽度为  $s \times w$ ，深度为  $d$ ，高度  $h$  为最高堆栈的高度。 $s$  个堆栈所需要的空间容量为  $s \times w \times d \times h$ 。由于  $h$  是第  $i$  至第  $j$  纸盒所构成的堆栈的高度(其中  $i \leq j$ )，因此  $h$  的可能取值可由  $n \times n$  矩阵  $H$  给出，其中对于  $i > j$  有  $H(i, j) = 0$ 。即有  $h = \sum_{k=i}^j h_k$ ， $i \leq j$ 。由于每个纸盒的高度可以认为是大于0，所以  $H(i, j) = 0$  代表一个不可能的高度。图 4-6a 给出了一个五个纸盒的堆栈。每个矩形中的数字代表纸盒的高度。图 4-6b 给出了五个纸盒堆栈折叠成三个堆栈后的情形，其中最大堆栈的高度为7。矩阵  $H$  是一个上三角矩阵，如图 4-6c 所示。

### 4.3.2 对角矩阵

可以采用如下所示的二维数组来描述一个元素类型为  $T$  的  $n \times n$  对角矩阵  $D$ ：

$T \text{ d}[n][n]$

使用数组元素  $d[i-1][j-1]$  来表示矩阵元素  $D(i, j)$ ，这种描述形式所需要的存储空间为  $n^2 \times \text{sizeof}(T)$ 。由于一个对角矩阵最多包含  $n$  个非0元素，所以可以采用如下所示的一维数组来描述对角矩阵：

$T \text{ d}[n]$

使用数组元素  $d[i-1]$  来表示矩阵元素  $D[i, i]$ 。根据对角矩阵的定义，所有未在一维数组中出现的矩阵元素均为0。可以采用程序 4-17 所示的 C++ 类 `DiagonalMatrix` 来实现这种描述。

程序 4-17 `DiagonalMatrix` 类

---

```
template<class T>
class DiagonalMatrix {
public:
    DiagonalMatrix(int size = 10)
        {n = size; d = new T [n];}
    ~DiagonalMatrix() {delete [] d;} // 析构函数
    DiagonalMatrix<T>&
        Store(const T& x, int i, int j);
    T Retrieve(int i, int j) const;
private:
    int n; // 矩阵维数
    T *d; // 存储对角元素的一维数组
};

template<class T>
DiagonalMatrix<T>& DiagonalMatrix<T>::Store(const T& x, int i, int j)
// 把x存为 D(i,j).
{
    if (i < 1 || j < 1 || i > n || j > n)
        throw OutOfBounds();
    if (i != j && x != 0) throw MustBeZero();
    if (i == j) d[i-1] = x;
    return *this;
}

template <class T>
T DiagonalMatrix<T>::Retrieve(int i, int j) const
// 返回D(i,j).
{
    if (i < 1 || j < 1 || i > n || j > n)
```

---

```

        throw OutOfBounds();
    if (i == j) return d[i-1];
    else return 0;
}

```

对于存储和搜索操作，提供了两个不同的函数（而不是靠重载操作符（）来完成）。在存储一个值时，必须保证不会在非对角线位置放置一个非 0 值，而在搜索一个值，没有必要检查对角线以外的值，因此有必要对这两种情形区别对待。程序 4-17 中 Store 和 Retrieve 的复杂性均为  $\Theta(1)$ 。

### 4.3.3 三对角矩阵

在一个  $n \times n$  三对角矩阵  $T$  中，非 0 元素排列在如下的三条对角线上：

- 1) 主对角线—— $i = j$ 。
- 2) 主对角线之下的对角线（称低对角线）—— $i = j+1$ 。
- 3) 主对角线之上的对角线（称高对角线）—— $i = j-1$ 。

这三条对角线上的元素总数为  $3n-2$ ，故可以使用一个拥有  $3n-2$  个位置的一维数组来描述  $T$ ，因为仅需要存储三条对角线上的元素。考察图 4-4b 所示的  $4 \times 4$  三对角矩阵，三条对角线上共有 10 个元素。如果把这些元素逐行映射到  $t$  中，则有  $t[0:9] = [2, 1, 3, 1, 3, 5, 2, 7, 9, 0]$ ；如果逐列映射到  $t$  中，则有  $t = [2, 3, 1, 1, 5, 3, 2, 9, 7, 0]$ ；如果按照对角线的次序（从最下面的对角线开始）进行映射，则有  $t = [3, 5, 9, 2, 1, 2, 0, 1, 3, 7]$ 。正如我们所看到的，可以有三种不同的选择来进行  $T$  到  $t$  的映射。每一种映射方式都要求有相对应的 Store 和 Retrieve 函数。程序 4-18 定义了 C++ 类 TridiagonalMatrix，其中采用了对角线映射方式。

程序 4-18 TridiagonalMatrix 类

```

template<class T>
class TridiagonalMatrix {
public:
    TridiagonalMatrix(int size = 10)
        { n = size; t = new T [3*n-2]; }
    ~TridiagonalMatrix() { delete [] t; }
    TridiagonalMatrix<T>& Store (const T& x, int i, int j);
    T Retrieve(int i, int j) const;
private:
    int n; // 矩阵维数
    T *t; // 存储三对角矩阵的一维数组
};

template<class T>
TridiagonalMatrix<T>& TridiagonalMatrix<T>:: Store(const T& x, int i, int j)
{ // 把 x 存为 T(i,j)
    if ( i < 1 || j < 1 || i > n || j > n)
        throw OutOfBounds();
    switch (i - j) {
        case 1: // 低对角线
            t[i - 2] = x; break;
        case 0: // 主对角线

```

```

    t[n + i - 2] = x; break;
case -1: // 高对角线
    t[2 * n + i - 2] = x; break;
default: if(x != 0) throw MustBeZero();
}
return *this;
}
template <class T>
T TridiagonalMatrix<T>::Retrieve(int i, int j) const
{// 返回T(i,j)
    if ( i < 1 || j < 1 || i > n || j > n)
        throw OutOfBounds();

    switch (i - j) {
        case 1: //低对角线
            return t[i - 2];
        case 0: // 主对角线
            return t[n + i - 2];
        case -1: // 高对角线
            return t[2 * n + i - 2];
        default: return 0;
    }
}
}

```

#### 4.3.4 三角矩阵

在一个三角矩阵中，非0元素都位于图4-7所示的“非0”区域。在一个下三角矩阵中，非0区域的第一行有1个元素，第二行有2个元素，…，第 $n$ 行有 $n$ 个元素；而在一个上三角矩阵中，非0区域的第一行有 $n$ 个元素，第二行有 $n-1$ 个元素，…，第 $n$ 行有1个元素。对于这两种情况，非0区域的元素总数均为：

$$\sum_{i=1}^n i = n(n+1)/2$$

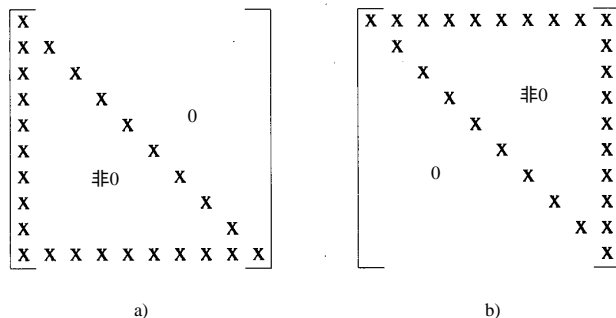


图4-7 下三角矩阵和上三角矩阵

a) 下三角矩阵 b) 上三角矩阵

两种三角矩阵都可以用一个大小为  $n(n+1)/2$  的一维数组来进行描述。考察把一个下三角矩阵  $L$  映射到一个一维数组  $l$ ，可采用按行和按列两种不同的方式进行映射。如果按行的方式进行映射，则对于图4-4c 的  $4 \times 4$  下三角矩阵可得到  $l[0:9] = (2, 5, 1, 0, 3, 1, 4, 2, 7, 0)$ ；若按列的方式进行映射，则得到  $l = (2, 5, 0, 4, 1, 3, 2, 1, 7, 0)$ 。

考察一个下三角矩阵中的元素  $L(i, j)$ 。如果  $i < j$ ，则  $L(i, j) = 0$ ；如果  $i \geq j$ ，则  $L(i, j)$  位于非0区域。在按行映射方式中，在元素  $L(i, j)$  之前共有  $\sum_{k=1}^{i-1} k + j - 1 = i(i-1)/2 + j - 1$  个元素位于非0区域，这个表达式同时给出了  $L(i, j)$  在  $l$  中的位置。采用这个公式，可得到程序4-19所示的Store和Retrieve函数，二者的时间复杂性均为  $\Theta(1)$ 。

程序4-19 LowerMatrix类

---

```
template<class T>
class LowerMatrix {
public:
    LowerMatrix(int size = 10)
        {n = size; t = new T [n*(n+1)/2];}
    ~LowerMatrix() {delete [] t;}
    LowerMatrix<T>& Store(const T& x, int i, int j);
    T Retrieve(int i, int j) const;
private:
    int n; // 矩阵维数
    T *t; // 存储下三角矩阵的一维数组
};

template<class T>
LowerMatrix<T>& LowerMatrix<T>:: Store(const T& x, int i, int j)
// 把x 存为 L(i,j).
{
    if ( i < 1 || j < 1 || i > n || j > n)
        throw OutOfBounds();
    // 当且仅当 i ≥ j 时(i,j) 位于下三角
    if (i >= j) t[i*(i-1)/2+j-1] = x;
    else if (x != 0) throw MustBeZero();
    return *this;
}

template <class T>
T LowerMatrix<T>::Retrieve(int i, int j) const
//返回 L(i,j).
{
    if ( i < 1 || j < 1 || i > n || j > n)
        throw OutOfBounds();
    // 当且仅当 i ≥ j 时(i,j) 位于下三角
    if (i >= j) return t[i*(i-1)/2+j-1];
    else return 0;
}
```

---

#### 4.3.5 对称矩阵

一个  $n \times n$  的对称矩阵可以用一个大小为  $n(n+1)/2$  的一维数组来描述，可参考三角矩阵的存储模式来存储矩阵的上三角或下三角。可以根据已存储的元素来推算出未存储的元素。

## 练习

20. 1) 扩充DiagonalMatrix类 (见程序4-17), 增加以下共享成员: 输入、输出、加、减、乘和矩阵转置函数。注意每种函数所得到的结果是一个用一维数组表示的对角矩阵。重载操作符 $<<$ ,  $>>$ ,  $+$ ,  $-$  和  $*$ 。

2) 试测试代码。

3) 每个函数的时间复杂性是多少?

21. 1) 扩充TridiagonalMatrix类 (见程序4-18), 增加以下共享成员: 输入、输出、加、减、乘和矩阵转置函数。重载适当的C++操作符。

2) 试测试代码。

3) 每个函数的时间复杂性是多少?

22. 1) 设计一个C++类TriByCols, 它把一个 $n \times n$ 的三对角矩阵按列的方式映射到一个大小为 $3n-2$ 的一维数组。设置以下共享成员: 输入、输出、存储、搜索、加、减和矩阵转置函数。

2) 试测试代码。

3) 每个函数的时间复杂性是多少?

23. 对于类TriByRows完成练习22, 它把一个 $n \times n$ 的三对角矩阵按行的方式映射到一个大小为 $3n-2$ 的一维数组。

24. 两个三对角矩阵的乘积仍然是一个三对角矩阵吗?

25. 对于一个上三角矩阵, 模仿程序4-19设计一个C++类UpperMatrix。

26. 扩充类LowerMatrix, 增加一个共享成员函数, 该函数可用来执行两个下三角矩阵的加法运算。函数的时间复杂性是多少?

27. 编写一个函数, 该函数能把一个下三角矩阵 (类型为LowerMatrix) 转置成一个上三角矩阵 (类型为UpperMatrix)。函数的时间复杂性是多少?

28. 如图4-8所示, 在一个 $n \times n$ 的C-矩阵中, 除第一行、第 $n$ 行和第一列外, 其他的元素均为0。一个C-矩阵最多有 $3n-2$ 个非0项。可把一个C-矩阵压缩存储到一个一维数组, 方法是首先存储第一行, 然后是第 $n$ 行, 最后是第一列中的剩余元素。

1) 给出一个 $4 \times 4$ 的C-矩阵样例及其压缩存储格式。

2) 按照上述思想设计一个C++类CMatrix, 它用一个一维数组 $t$ 描述一个 $n \times n$ 的C-矩阵, 要求提供两个共享成员函数Store和Retrieve。

3) 使用适当的测试数据来测试代码。

29. 编写一个对两个下三角矩阵 (类型为LowerMatrix) 进行乘法运算的函数, 所得到的结果用一个二维数组来描述。函数的时间复杂性是多少?

30. 编写函数, 对一个下三角矩阵和一个上三角矩阵进行乘法运算 (两个矩阵都是按行的方式存储在一个一维数组中), 所得到的结果用一个二维数组来描述。函数的时间复杂性是多少?

31. 假定按行的方式把对称矩阵的下三角区域存储在一个一维数组中。试设计一个C++类LowSymmetric, 要求提供两个共享成员函数Store和Retrieve, 这两个函数的时间复杂性应为 $\Theta(1)$ 。

X	X	X	X	X	X	X
X						
X						
X						
X						
X						
X	X	X	X	X	X	X

x表示可能为非0  
所有其他项均为0

图4-8 C-矩阵

32. 令 $A$ 和 $B$ 是两个 $n \times n$ 下三角矩阵。两个矩阵的下三角区域中的元素总数为 $n(n+1)$ 。设计一种存储模式，用以在一个数组 $d[n+1][n]$ 中同时存储这两个下三角区域。(提示：如果把 $A$ 的下三角区域与 $B^T$ 的上三角区域进行合并，就可以得到一个 $(n+1) \times n$ 的矩阵)。对于 $A$ 和 $B$ ，分别给出其相应的存储函数和搜索函数。每个函数的复杂性应为 $\Theta(1)$ 。

\*33. 带状方阵 (square band matrix)  $D_{n,a}$  是一个 $n \times n$ 的矩阵，其中所有非0元素都落在包围主对角线的带状区域中。如图4-9所示，带状区域包含主对角线以及主对角线两边的 $a-1$ 条对角线。

- 1) 在 $D_{n,a}$ 的带状区域中有多少个元素？
- 2) 对于 $D_{n,a}$ 带状区域中的元素 $d_{ij}$ 来说， $i$ 和 $j$ 之间应满足什么关系？
- 3) 假定按对角线的方式把 $D_{n,a}$ 的带状区域映射到一个一维数组 $b$ 中，从最下面的一条对角线开始映射。例如，图4-9中的带状方阵 $D_{4,3}$ 可按如下形式来存储：

b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	b[8]	b[9]	b[10]	b[11]	b[12]	b[13]
9	7	8	3	6	6	0	2	8	7	4	9	8	4
$d_{20}$	$d_{31}$	$d_{10}$	$d_{21}$	$d_{32}$	$d_{00}$	$d_{11}$	$d_{22}$	$d_{33}$	$d_{01}$	$d_{12}$	$d_{23}$	$d_{02}$	$d_{13}$

给出一个公式，用来计算带状区下方元素 $d_{ij}$ 的存储位置（在上例中 $d_{10}$ 的位置为2）。

- 4) 使用3)中的映射方法设计一个C++类SquareBand，要求提供两个共享成员函数Store和Retrieve，这两个函数的时间复杂性分别是多少？

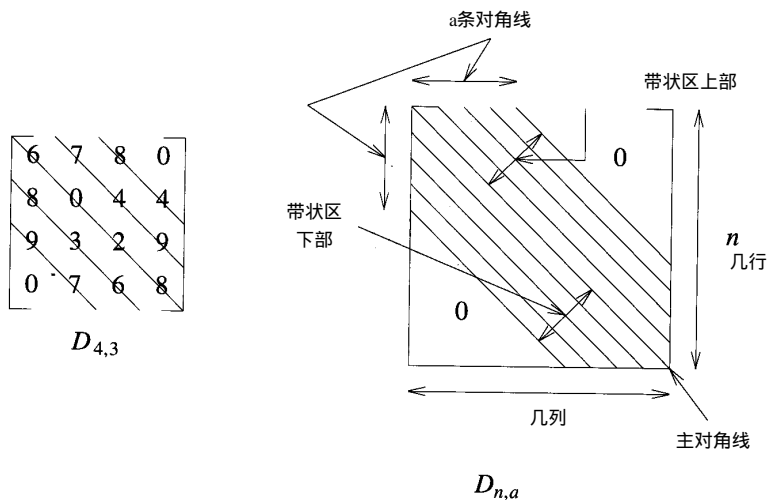


图4-9 带状方阵

34. 一个 $n \times n$ 的矩阵 $T$ 是一个等对角矩阵 (Toeplitz matrix) 当且仅当对于所有的 $i$ 和 $j$ 有 $T(i,j)=T(i-1,j-1)$ ，其中 $i>1, j>1$ 。

- 1) 证明一个等对角矩阵最多有 $2n-1$ 个不同的元素。
- 2) 给出一种映射模式，把一个等对角矩阵映射到一个大小为 $2n-1$ 的一维数组中。
- 3) 采用2)中的映射模式设计一个C++类Toeplitz，用一个大小为 $2n-1$ 的一维数组来存储等对角矩阵。要求给出两个共享成员函数Store和Retrieve，这两个函数的时间复杂性应为 $\Theta(1)$ 。
- 4) 编写一个共享成员函数，用来对两个按2)中所给方式存储的等对角矩阵进行乘法运算，

所得到的结果存储在一个二维数组中。该函数的时间复杂性是多少？

35. 一个  $n \times n$  的矩阵  $M$  是一个反对角矩阵 (antidiagonal) 当且仅当对于所有满足  $i+j = n+1$  的  $i$  和  $j$  有  $M(i, j) \neq 0$ 。

- 1) 给出一个  $4 \times 4$  反对角矩阵的样例。
- 2) 证明反对角矩阵  $M$  最多有  $n$  个非 0 元素。
- 3) 设计一种映射模式，用来把一个反对角矩阵映射到一个大小为  $n$  的一维数组之中。
- 4) 采用 3) 中的映射模式设计一个 C++ 类 AntiDiagonal，要求给出两个共享成员函数 Store 和 Retrieve。
- 5) Store 和 Retrieve 的时间复杂性分别是多少？

## 4.4 稀疏矩阵

### 4.4.1 基本概念

如果一个  $m \times n$  矩阵中有“许多”元素为 0，则称该矩阵为稀疏矩阵 (sparse)。不是稀疏的矩阵被称为稠密矩阵 (dense)。在稀疏矩阵和稠密矩阵之间并没有一个精确的界限。 $n \times n$  的对角矩阵和三对角矩阵都是稀疏矩阵，二者都有  $O(n)$  个非 0 元素和  $O(n^2)$  个 0 元素。一个  $n \times n$  的三角矩阵是稀疏矩阵吗？它至少有  $n(n-1)/2$  个 0 元素，最多有  $n(n+1)/2$  个非 0 元素。在本节中我们规定若一个矩阵是稀疏矩阵，则其非 0 元素的数目应小于  $n^2/3$ ，在有些情况下应小于  $n^2/5$ ，因此可将三角矩阵视为稠密矩阵。

诸如对角矩阵和三对角矩阵这样的稀疏矩阵，其非 0 区域的结构很有规律，因此可以设计一个很简单的存储结构，该存储结构的大小就等于矩阵非 0 区域的大小。本节中主要考察具有不规则非 0 区域的稀疏矩阵。

例 4-5 某超级市场正在开展一项关于顾客购物品种的研究。为了完成这项研究，收集了 1000 个顾客的购物数据，这些数据被组织成一个矩阵  $purchases$ ，其中  $purchases(i, j)$  表示顾客  $j$  所购买商品  $i$  的数量。假定该超级市场有 10 000 种不同的商品，那么  $purchases$  将是一个  $10\,000 \times 1000$  的矩阵。如果每个顾客平均购买了 20 种不同商品，那么在 10 000 000 个矩阵元素将大约只有 20 000 个元素为非 0，并且非 0 元素的分布没有很明确的规律。

超级市场有一个  $10\,000 \times 1$  的价格矩阵  $price$ ， $price(i)$  代表商品  $i$  的单价。矩阵  $spent = purchases^T * price$  是一个  $1000 \times 1$  的矩阵，它给出每个顾客所花费的购物资金。如果用一个二维数组来描述矩阵  $purchases$ ，那么将浪费大量的存储空间，并且计算  $spent$  时也将耗费更多的时间。

### 4.4.2 数组描述

可以按行主次序把不规则稀疏矩阵映射到一维数组中。例如图 4-10a 中的  $4 \times 8$  矩阵按行主次序进行存储可得到：2, 1, 6, 7, 3, 9, 8, 4, 5。

为了重建矩阵结构，必须记录每个非 0 元素所在的行号和列号，所以在把稀疏矩阵的非 0 元素映射到数组中时必须提供三个域：row (矩阵元素所在行号)、col (矩阵元素所在列号) 和 value (矩阵元素的值)。为此，定义如下所示的模板类 Term：

```
template <class T>
class Term {
private:
```



```
int row, col;
T value;
};
```

如果a是一个类型为Term的数组，那么图4-10a中的稀疏矩阵按行主次序存储到a中所得到的结果如图4-10b所示。除了存储数组a以外，还必须存储矩阵行数、矩阵列数和非0项的数目。所以存储图4-10a中的九个非0元素所需要的存储器字节数为 $21 * \text{sizeof}(\text{int}) + 9 * \text{sizeof}(\text{T})$ 。如果用 $4 \times 8$ 的数组来描述这个矩阵，则需要的字节数为 $32 * \text{sizeof}(\text{T})$ 。假定T是int类型且 $\text{sizeof}(\text{T})=2$ ，那么图4-10b中的描述需60个字节，而采用 $4 \times 8$ 的数组则需要64个字节。对于这个例子用一维数组来进行存储并没有节省出多少空间。不过，对于例4-5中的矩阵purchase，其一维数组描述需 $60\,000 * \text{sizeof}(\text{T})$ 个字节，而二维数组描述则需 $10\,000\,000 * \text{sizeof}(\text{T})$ 个字节。如果一个整数占2个字节，则节省的空间为19 880 000字节。

```
0 0 0 2 0 0 1 0
0 6 0 0 7 0 0 3
0 0 0 9 0 8 0 0
0 4 5 0 0 0 0 0
```

a)

a[]	0	1	2	3	4	5	6	7	8
row	1	1	2	2	2	3	3	4	4
col	4	7	2	5	8	4	6	2	3
value	2	1	6	7	3	9	8	4	5

b)

图4-10 稀疏矩阵及其数组描述

a)  $4 \times 8$ 矩阵 b) 数组描述

稀疏矩阵的数组描述并不能使Store和Retrieve函数的执行效率也得到相应提高。Store函数所需要的时间为 $O(\text{非0元素数目})$ ，因为可能需要移动这么多元素以便为新元素留出空间。如果采用折半搜索，则Retrieve函数所需要的时间为 $O(\log[\text{非0元素数目}])$ 。如果采用标准的二维数组来描述矩阵，这两种函数所需要的时间均为 $\Theta(1)$ 。不过，诸如转置、加和乘这样的矩阵操作可以得到高效地执行。

### 1. 类SparseMatrix

可以定义一个类SparseMatrix（见程序4-20），用来把稀疏矩阵按行主次序映射到一维数组中。这个类是Term的友元。私有成员rows、cols和terms代表稀疏矩阵中的行数、列数和非0元素的数目。MaxTerms是数组a（用来存储非0元素）的容量。在定义共享成员时，没有定义加法操作符+，因为它会创建一个临时结果，这个临时结果必须复制到所返回的环境才可以使用。由于SparseMatrix的复制构造函数将会复制每一个元素，因此操作符+中的复制代价太大，如使用Add函数则可以避免，它将会告诉你把结果送到哪里去。

程序4-20 类SparseMatrix

```
template<class T>
class SparseMatrix
{
    friend ostream& operator<< (ostream&, const SparseMatrix<T>&);
    friend istream& operator>> (istream&, SparseMatrix<T>&);
public:
    SparseMatrix(int maxTerms = 10);
    ~SparseMatrix() {delete [] a;}
    void Transpose(SparseMatrix<T> &b) const;
```

```

void Add(const SparseMatrix<T> &b, SparseMatrix<T> &c) const;
private:
void Append(const Term<T>& t);
int rows, cols; //矩阵维数
int terms; // 非0元素数目
Term<T> *a; // 存储非0元素的数组
int MaxTerms; // 数组a的大小;
};

```

程序4-21给出了SparseMatrix的构造函数，程序4-22 给出了输入操作符<< 和输出操作符>>的代码，其中operator<<和operator>>必须是类Term的友元。operator>>按行主次序输出稀疏矩阵的非0元素，而operator<<则按行主次序输入稀疏矩阵并建立内部数组描述。operator<<和operator>>的时间复杂性均为 $\Theta(\text{terms})$ 。练习37和38要求对程序4-22中的代码进行细化。

程序4-21 类SparseMatrix的构造函数

```

template<class T>
SparseMatrix<T>::SparseMatrix(int maxTerms)
{ // 稀疏矩阵的构造函数
    if (maxTerms < 1) throw BadInitializers();
    MaxTerms = maxTerms;
    a = new Term<T> [MaxTerms];
    terms = rows = cols = 0;
}

```

程序4-22 类SparseMatrix的输入和输出函数

```

// 重载<<
template <class T>
ostream& operator<<(ostream& out, const SparseMatrix<T>& x)
{ // 把*this 送至输出流
    // 输出矩阵的特征
    out << "rows = " << x.rows << " columns = " << x.cols << endl;
    out << "nonzero terms = " << x.terms << endl;
    // 输出非0元素，每行1个
    for (int i = 0; i < x.terms; i++)
        out << "a(" << x.a[i].row << ', ' << x.a[i].col << ") = " << x.a[i].value << endl;
    return out;
}

// 重载>>
template<class T>
istream& operator>>(istream& in, SparseMatrix<T>& x)
{ // 输入一个稀疏矩阵
    // 输入矩阵的特征
    cout << "Enter number of rows, columns, and terms" << endl;
    in >> x.rows >> x.cols >> x.terms;
    if (x.terms > x.MaxTerms) throw NoMem();
    // 输入矩阵元素
}

```

```
for (int i = 0; i < x.terms; i++) {
    cout << "Enter row, column, and value of term " << (i + 1) << endl;
    in >> x.a[i].row >> x.a[i].col >> x.a[i].value;
}
return in;
}
```

如果输入的元素数目超出了数组 \*this.a 的大小，则 operator>> 将引发一个异常。一种处理异常的方法是删除数组 a，然后使用 new 重新分配一个更大的数组。

## 2. 矩阵转置

程序4-23给出了函数 Transpose的代码。转置后的矩阵被返回到 b 中。首先验证 b 中是否有足够的空间来存储被转置矩阵的非 0 元素。如果空间不足，要么重新分配一个更大的数组 b.a，要么引发一个异常。在此程序中引发了一个异常。如果 b 中有足够的空间来容纳转置矩阵，则创建两个数组 ColSize 和 RowNext。ColSize[i] 是指矩阵第 i 列中的非 0 元素数，RowNext[i] 代表转置矩阵第 i 行的下一个非 0 元素在 b 中的位置。对 ColSize 的计算是在前两个 for 循环中通过简单地检查每个矩阵元素来完成的。对 RowNext 的计算是在第三个 for 循环中完成的。RowNext[i] 的初值为转置矩阵中第 0 行至第 i-1 列中的元素数目（即原矩阵中第 0 列至第 i-1 列中的元素数目）。在最后一个 for 循环中，非 0 元素被复制到 b 中相应位置。

程序4-23 转置一个稀疏矩阵

```
template<class T>
void SparseMatrix<T>:: Transpose(SparseMatrix<T> &b) const
{
    // 把 *this 的转置结果送入 b
    // 确信 b 有足够的空间
    if (terms > b.MaxTerms) throw NoMem();
    // 设置转置特征
    b.cols = rows;
    b.rows = cols;
    b.terms = terms;
    // 初始化
    int *ColSize, *RowNext;
    ColSize = new int[cols + 1];
    RowNext = new int[rows + 1];
    // 计算 *this 每一列的非 0 元素数
    for (int i = 1; i <= cols; i++) // 初始化
        ColSize[i] = 0;
    for (int = 0; i < terms; i++)
        ColSize[a[i].col]++;
    // 给出 b 中每一行的起始点
    RowNext[1] = 0;
    for (int i = 2; i <= cols; i++)
        RowNext[i] = RowNext[i - 1] + ColSize[i - 1];

    // 执行转置操作
    for (int i = 0; i < terms; i++) {
        int j = RowNext[a[i].col]++; // 在 b 中的位置
```

```

        b.a[j].row = a[i].col;
        b.a[j].col = a[i].row;
        b.a[j].value = a[i].value;
    }
}

```

尽管程序 4-23 比按二维数组存储矩阵（见程序 2-22）更复杂，但对于有很多个 0 元素的矩阵，程序 4-23 要快得多。不难发现，对于例 4-5 中的矩阵 purchase，采用一维数组描述来完成转置操作要比按二维数组描述完成转置操作更快。Transpose 的时间复杂性为  $O(\text{cols} + \text{terms})$ 。

### 3. 两个矩阵相加

在两个矩阵相加的代码中使用了程序 4-24 中的函数 Append，它把一个非 0 项添加到一个稀疏矩阵的非 0 项数组的尾部。该函数的时间复杂性为  $\Theta(1)$ 。

程序 4-24 添加一个非 0 元素

```

template<class T>
void SparseMatrix<T>::Append(const Term<T>& t)
{// 把一个非 0 元素 t 添加到 *this 之中
    if (terms >= MaxTerms) throw NoMem();
    a[terms] = t;
    terms++;
}

```

程序 4-25 中的代码对两个矩阵 \*this 和 b 执行加法操作，所得到的结果放入 c 中。矩阵 c 的获得是通过从左至右依次扫描两个矩阵中的元素来实现的。在扫描过程中使用了两个游标：ct（矩阵 \*this 的游标）和 cb（矩阵 b 的游标）。在每一次 while 循环过程中，需要确定 (\*this).a[ct] 的位置或是在 b.a[cb] 之前，或是相同，或是在 b.a[cb] 之后。判断的方法是分别计算出这两项的索引（行主次序）。在 Add 函数中，\*this 的元素索引由 indt 给出，b 的元素索引由 indb 给出。

程序 4-25 两个稀疏矩阵相加

```

template<class T>
void SparseMatrix<T>::Add(const SparseMatrix<T> &b, SparseMatrix<T> &c) const
{// 计算 c = (*this) + b.
    // 验证可行性
    if (rows != b.rows || cols != b.cols)
        throw SizeMismatch(); // 不能相加
    // 设置结果矩阵 c 的特征
    c.rows = rows;
    c.cols = cols;
    c.terms = 0; // 初值
    // 定义 *this 和 b 的游标
    int ct = 0, cb = 0;
    // 在 *this 和 b 中遍历
    while (ct < terms && cb < b.terms) {
        // 每一个元素的行主索引
        int indt = a[ct].row * cols + a[ct].col;
        int indb = b.a[cb].row * cols + b.a[cb].col;
    }
}

```

```

if (indt < indb) { // b 的元素在后
    c.Append(a[ct]);
    ct++; } // *this的下一个元素
else {if (indt == indb) { // 位置相同
    //仅当不为0时才添加到 c中
    if (a[ct].value + b.a[cb].value) {
        Term < T;
        t.row = a[ct].row;
        t.col = a[ct].col;
        t.value = a[ct].value + b.a[cb].value;
        c.Append(t);
        ct++; cb++; } // *this 和 b的下一个元素
    else {c.Append(b.a[cb]); cb++; } // b的下一个元素
    }
}
// 复制剩余元素
for (; ct < terms; ct++)
    c.Append(a[ct]);
for (; cb < b.terms; cb++)
    c.Append(b.a[cb]);
}

```

函数Add中的while循环最多会执行  $\text{terms} + \text{b.terms}$  次，因为在循环过程中  $\text{ct}, \text{cb}$  会不断增值。第一个for循环最多会执行  $\text{terms}$  次，而第二个for循环最多会执行  $O(\text{b.terms})$  次。另外，每个循环中的每一次循环只需要常量时间。因此函数 Add的时间复杂性为  $O(\text{terms} + \text{b.terms})$ 。如果用二维数组来描述每个矩阵，则两个矩阵相加需耗时  $O(\text{rows} * \text{cols})$ 。当  $\text{terms} + \text{b.terms}$  远小于  $\text{rows} * \text{cols}$  时，稀疏矩阵的加法执行效率将大大提高。

#### 4.4.3 链表描述

用一维数组来描述稀疏矩阵所存在的缺点是：当我们创建这个一维数组时，必须知道稀疏矩阵中的非0元素总数。虽然在输入矩阵时这个数是已知的，但随着矩阵加法、减法和乘法操作的执行，非0元素的数目会发生变化，因此如果不实际计算，很难精确地知道非0元素的数目。正因为如此，只好先估计出每个矩阵中的非0元素数目，然后用它作为数组的初始大小。在设计SparseMatrix类时就采用了这样的策略。在该类的代码中，当结果矩阵非0元素的数目超出所估计的数目时将引发一个异常。不过也可以重写这些代码，在非0元素的数目超出所估计的数目时分配一个新的、更大的数组，然后从老数组中把元素复制出来并删除老数组。这种额外工作将使算法的效率降低，并留下了新数组到底应该取多大的问题。如果新数组不够大，还得重复上述分配和复制过程；如果新数组太大，又会浪费很多空间。一种解决办法就是采用基于指针的描述。这种方法需要额外的指针存储空间，但可以节省对数组描述中其他一些信息的存储。最重要的是，它可以避免存储的再分配以及部分结果的复制。

##### 1. 描述

链表描述的一种可行方案是把每行的非0元素串接在一起，构成一个链表，如图4-11所示。图中每个非阴影节点代表稀疏矩阵中的一个非0元素，它有三个域： $\text{col}$ （非0元素所在列号）、 $\text{value}$ （非0元素的值）和 $\text{link}$ （指向下一个非阴影节点的指针）。仅当矩阵某行中至

少包含一个非0元素才会为该行创建一个链表。在行链表中，每个节点按其 col值的升序进行排列。

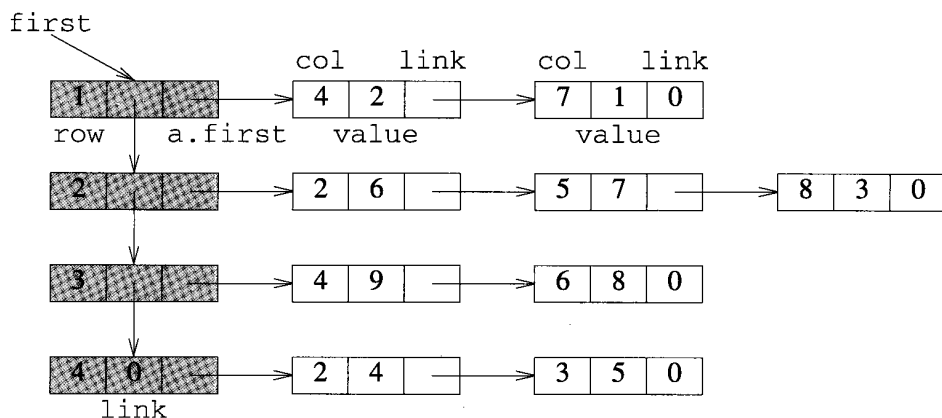


图4-11 图4-10a 中矩阵的链表描述

用另外一个链表把所有的行链表（即非阴影链表）收集在一起，如图 4-11中的阴影节点所示。每个阴影节点也有三个域：row（相应行链表的行号）、link（指向下一个阴影节点的指针）和a（指向行链表，a.first 指向行链表中的第一个节点）。各阴影节点按其 row值的升序排列。每个阴影节点可以被视为一个行链表的头节点，因此阴影链表可以被视为头节点链表。空的头节点链表代表没有非0元素的矩阵。

## 2. 链表节点类型

图4-11中非阴影节点所对应的链表可以被定义为 `Chain<CNode<T>>` 类型，其中 `CNode` 的定义见程序 4-26。阴影节点所对应的链表可以被定义为 `Chain<HeadNode<T>>` 类型，其中 `HeadNode<T>` 的定义见程序 4-26。

程序4-26 稀疏矩阵描述中所使用的链表节点

```
template<class T>
class CNode {
public:
    int operator !=(const CNode<T>& y)
    {return (value != y.value);}
    void Output(ostream& out) const
    {out << "column " << col << " value " << value;}
private:
    int col;
    T value;
};

template<class T>
ostream& operator<<(ostream& out, const CNode<T>& x)
    {x.Output(out); out << endl; return out;}

template<class T>
class HeadNode {
```

```

public:
    int operator !=(const HeadNode<T>& y)
    {return (row != y.row);}
    void Output(ostream& out) const
    {out << "row " << row;}
private:
    int row;
    Chain<CNode<T>> a; //行链表
};
template<class T>
ostream& operator<<(ostream& out, const HeadNode<T>& x)
{x.Output(out); out << endl; return out;}

```

### 3. 类LinkedMatrix

现在可以来定义类LinkedMatrix，见程序4-27。其中共享函数的实现以及操作符>>和<<的重载都要求LinkedMatrix，operator>>和operator<<是类CNode和HeadNode的友元。

程序4-27 用链表描述稀疏矩阵的类定义

```

template<class T>
class LinkedMatrix
{
    friend ostream& operator<<
    (ostream&, const LinkedMatrix<T>&);
    friend istream& operator>>
    (istream&, LinkedMatrix<T>&);
public:
    LinkedMatrix(){}
    ~LinkedMatrix(){}
    void Transpose(LinkedMatrix<T> &b) const;
    void Add(Const LinkedMatrix<T> &b, LinkedMatrix<T> &c) const;
private:
    int rows, cols; // 矩阵维数
    Chain<HeadNode<T>> a; // 头节点链表
};

```

### 4. 重载>>

程序4-28按行主次序输入所有的非0元素并创建图4-11所示的链表。它首先要求输入矩阵的维数以及非0元素的个数，然后输入各个非0元素并把它们收集到各行链表中。用变量H代表当前行链表的头节点。如果下一个非0元素不属于当前行链表，则将当前行链表添加到矩阵x的头节点链表x.a之中（虚设的第0行链表除外）。接下来，H被设置为指向一个新的行链表，同时将刚才那个非0元素添加到这个新的行链表之中。如果新的非0元素属于当前行链表，则只需简单地把它添加到链表H.a中即可。注意函数Append和Zero是3.4.3节所定义的扩充链表类的成员。Append在链表的尾部添加一个节点，而Zero则把first置为0但并不删除链表中的节点。operator>>的复杂性为O(terms)。



程序4-28 输入一个稀疏矩阵

```

template<class T>
istream& operator>>(istream& in, LinkedMatrix<T>& x)
{
    // 从输入流中输入矩阵 x
    x.a.Erase(); // 删除x中的所有节点
    // 获取矩阵特征
    int terms; // 输入的元素数
    cout << "Enter number of rows, columns, and terms" << endl;
    in >> x.rows >> x.cols >> terms;
    // 虚设第0行
    HeadNode<T> H; // 当前行的头节点
    H.row = 0; // 当前行号
    // 输入 x的非0元素
    for (int i = 1; i <= terms; i++) {
        // 输入下一个元素
        cout << "Enter row, column, and value of term " << i << endl;
        int row, col;
        T value;
        in >> row >> col >> value;
        // 检查新元素是否属于当前行
        if (row > H.row) { // 开始一个新行
            // 如果不是第0行, 则把当前行的头节点 H添加到头节点链表x.a之中
            if (H.row) x.a.Append(H);
            // 为新的一行准备 H
            H.row = row;
            H.a.Zero(); // 置链表头指针 first=0
        }
        // 添加新元素
        CNode<T> *c = new CNode<T>;
        c->col = col;
        c->value = value;
        H.a.Append(*c);
    }
    // 注意矩阵的最后一行
    if (H.row) x.a.Append(H);
    H.a.Zero(); // 置链表头指针为 0
    return in;
}

```

### 5. 重载<<

为了输出用链表表示的稀疏矩阵, 使用一个链表遍历器(见 3.4.4节)依次检查头节点链表中的每个节点。对于头节点链表中的每个节点, 输出其相应的行链表。程序 4-29给出了操作符<<的实现代码。代码的时间复杂性与非0元素的数目呈正比。

程序4-29 输出一个稀疏矩阵

```

template<class T>
ostream& operator<<(ostream& out, const LinkedMatrix<T>& x)

```

```

// 把矩阵 x 送至输出流
ChainIterator<HeadNode<T> > p; // 头节点遍历器
// 输出矩阵的维数
out << "rows = " << x.rows << " columns = " << x.cols << endl;
// 将 h 指向第一个头节点
HeadNode<T> *h = p.Initialize(x.a);
if (!h) {out << "No non-zero terms" << endl;
        return out;}
// 每次输出一行
while (h) {
    out << "row " << h->row << endl;
    out << h->a << endl; //输出行链表
    h = p.Next();        // 下一个头节点
}
return out;
}

```

## 6. 函数Transpose

对于转置操作，可以采用箱子来从矩阵 \*this 中收集位于结果矩阵同一行的非0元素。bin[i] 是结果矩阵b 中第i 行非0元素所对应的链表。在程序 4-30的while 循环中，沿着输入矩阵 \*this 的头节点链表按行主次序依次检查每个非 0元素（在每个行链表中按从左至右的次序检查）。用链表遍历器 p 来遍历头节点链表，用链表遍历器 q 遍历行链表。在按照上述次序遍历 \*this 的过程中，把遇到的每个非0元素添加到相应的箱子链表中。最后用一个for循环来收集各箱子链表，并创建结果矩阵所需要的头节点链表。

程序4-30 转置一个稀疏矩阵

```

template<class T>
void LinkedMatrix<T>::
    Transpose(LinkedMatrix<T> &b) const
{
    // 转置 *this，并把结果放入b
    b.a.Erase(); // 删除b中所有节点
    // 创建用来收集b中各行元素的箱子
    Chain<CNode<T> > *bin;
    bin = new Chain<CNode<T> > [cols + 1];
    // 头节点遍历器
    ChainIterator<HeadNode<T> > p;
    // h 指向*this的第一个头节点
    HeadNode<T> *h = p.Initialize(a);
    // 把 *this的元素复制到箱子中
    while (h) { // 检查所有的行
        int r = h->row; // 行链表中的行数
        // 行链表遍历器
        ChainIterator<CNode<T> > q;
        // z指向行链表的第一个节点
        CNode<T> *z = q.Initialize(h->a);
        CNode<T> x; // 临时节点
        // *this第r行中的元素变成b中第r列的元素
    }
}

```

```

x.col = r;
//检查 *this中第r行的所有非0元素
while (z) { // 遍历第 r行
    x.value = z->value;
    bin[z->col].Append(x);
    z = q.Next(); // 该行的下一个元素
}
h = p.Next(); // 继续下一行
}
// 设置b的维数
b.rows = cols;
b.cols = rows;
// 装配 b的头节点链表
HeadNode<T> H;
// 搜索箱子
for (int i = 1; i <= cols; i++)
    if (!bin[i].IsEmpty()) { // 转置矩阵的第 i 行
        H.row = i;
        H.a = bin[i];
        b.a.Append(H);
        bin[i].Zero(); // 置链表头指针为0
    }
H.a.Zero(); // 置链表头指针为0
delete [] bin;
}

```

while循环所需要的时间与非0元素的数目呈线性关系，而for循环所需要的时间与输入矩阵的列数呈线性关系。因此总的时间与这两个量的和呈线性关系。

练习45要求你实现Add函数以及其他基本函数。

## 练习

36. 对于一个按行主次序存储在一维数组中的稀疏矩阵，编写其 Store和Retrive函数。函数的时间复杂性分别是多少？

37. 细化程序4-22中的operator>>，要求验证：是否按行主次序输入链表，每个非0元素的行号和列号是否有效，所输入的元素是否非0。

38. 修改程序4-22中的operator>>，要求如果 x.MaxSize<x.terms，则分配一个更大的数组来存储矩阵元素。

39. 编写类SparseMatrix的复制构造函数。

40. 修改程序4-24中的Append函数，要求如果当前数组a 没有足够的空间，则分配一个更大的数组a。

41. 扩充类SparseMatrix（见程序4-20），增加两个共享成员函数——矩阵加和矩阵乘。

42. 假定按列主次序把一个稀疏矩阵映射到一个一维数组中

1) 给出图4-10a 中稀疏矩阵的描述。

2) 对按照这种方式存储的稀疏矩阵，编写相应的Store和Retrieve函数。

3) Store和Retrieve函数的时间复杂性分别是多少？

\*43. 编写一个函数，对两个存储在一维数组中的稀疏矩阵进行乘法运算。假定两个矩阵都

是按行主次序存储的。所得到的结果也要求按行主次序存储。

\*44. 编写一个函数，对两个存储在一维数组中的稀疏矩阵进行乘法运算。假定两个矩阵都是按列主次序存储的。所得到的结果也要求按列主次序存储。

\*45. 通过为下列操作增加共享成员扩充类 LinkedMatrix：

- 1) 存储一个元素，已知行号、列号和元素的值。
- 2) 在矩阵中查找具有给定行号和列号的元素。
- 3) 两个稀疏矩阵相加。
- 4) 两个稀疏矩阵相减。
- 5) 两个稀疏矩阵相乘。

\*46. 可以采用如下所述的链表来描述稀疏矩阵。链表节点中包括如下域：down, right, row, col 和 value。稀疏矩阵的每个非0元素都用一个节点来表示。0元素不必存储。所有的节点链接在一起形成两个循环链表。第一个链表——行链表——使用 right 域按行的次序（每行按列号的次序）连接所有节点。第二个链表——列链表——使用 down 域按列的次序（每列按行号的次序）连接所有节点。这两个链表共享同一个头节点。此外，有一个附加的节点用来存储矩阵的维数。

1) 给出一个  $5 \times 8$  的矩阵，其中正好有9个非0元素，并且在每一行和每一列中至少有一个非0元素。对于这个稀疏矩阵，给出其相应的链表描述。

2) 假定一个  $m \times n$  矩阵中有  $t$  个非0元素，若按上述形式来描述该矩阵，则  $t$  应该有多小才能保证所需要的存储空间比使用一个  $m \times n$  数组要来得少？

3) 设计一个合适的外部描述（即可用来输入和输出），该描述不应要求输入0元素。

4) 采用2) 中的描述完成练习 37。

5) 对于类的每个共享成员，给出其时间复杂性。这些复杂性与采用二维数组来描述矩阵所得到的复杂性相比有哪些不同？

## 第5章 堆 栈

堆栈和队列可能是使用频率最高的数据结构，二者都来自于第 3 章中的线性表数据结构（经过某种限制以后）。本章将研究堆栈，下一章研究队列。堆栈数据结构是通过对线性表的插入和删除操作进行限制而得到的（插入和删除操作都必须在表的同一端完成），因此，堆栈是一个后进先出（last-in-first-out, LIFO）的数据结构。

由于堆栈是一种特殊的线性表，因此可以很自然地由相应的线性表类中派生出堆栈类。我们可以从程序 3-1 的 LinearList 类派生出基于公式描述的堆栈类，也可以从程序 3-8 的 Chain 类派生出基于链表结构的堆栈类。通过类的派生，可以大大简化程序设计的任务，但同时代码的执行效率有明显损失。由于堆栈是一个很基本的数据结构，许多程序都要用到堆栈，本章中也直接给出了基于公式描述和基于链表结构的堆栈类（而不是从其他类派生而来）。这两种类与对应的派生类相比，在执行效率上将有很大提高。

本章还给出六个使用堆栈的应用程序。第一个应用是一个用来匹配表达式中左、右括号的简单程序；第二个应用是一个经典的汉诺塔问题求解程序。汉诺塔问题要求把一个塔上的所有碟子按照一定的规则搬到另一个塔上，每次只能搬动一个碟子，其间可以借助于第 3 个塔的帮助。在汉诺塔问题的求解过程中，每个塔都被视为一个堆栈；第三个应用使用堆栈来解决火车车厢重排问题，其目标是把火车车厢按所希望的次序重新排列；第四个应用是关于电子布线的问题，在这个应用中，用堆栈来确定一个电路是否可以成功布线；第五个应用用于解决 3.8.3 节所介绍的离线等价类问题。采用堆栈可以帮助我们在线性时间内确定等价类；最后一个应用用来解决经典的迷宫问题，即寻找一条从入口到出口的迷宫路径。应该仔细地研究这个应用，因为它体现了许多软件工程的原理和思想。

本章中所用到的 C++ 语言的新的特性是派生类和继承。

### 5.1 抽象数据类型

定义 [堆栈] 堆栈（stack）是一个线性表，其插入（也称为添加）和删除操作都在表的同一端进行。其中一端被称为栈顶（top），另一端被称为栈底（bottom）。

图 5-1a 给出了一个四元素的堆栈。假定希望在 5-1a 的堆栈中添加一个元素 E，这个元素将被放到元素 D 的顶部，所得到的结果如图 5-1b 所示。如果想从 5-1b 的堆栈中删除一个元素，那么元素 E 将被删除，删除 E 之后又将得到 5-1a 的结果。如果对 5-1b 的堆栈连续执行三次删除操作，则将得到图 5-1c 所示的堆栈。

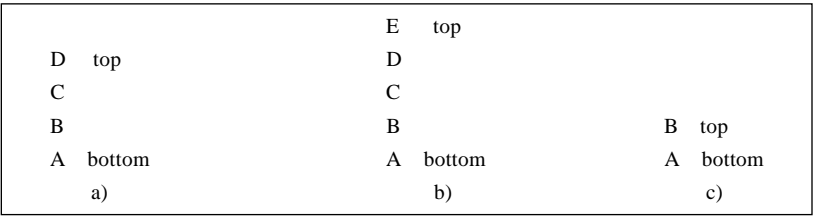


图 5-1 堆栈结构

从上面的讨论中可以看出，堆栈是一个后进先出表。这种类型的表在计算过程中将频繁使用。ADT堆栈的描述见ADT 5-1。

ADT5-1 堆栈的抽象数据类型描述

---

抽象数据类型 *Stack*{

实例

元素线性表，栈底，栈顶

操作

*Create()*：创建一个空的堆栈

*IsEmpty()*：如果堆栈为空，则返回 true，否则返回 false

*IsFull()*：如果堆栈满，则返回 true，否则返回 false

*Top()*：返回栈顶元素

*Add(x)*：向堆栈中添加元素 *x*

*Delete(x)*：删除栈顶元素，并将它传递给 *x*

}

---

## 5.2 派生类和继承

我们经常会处理这样一类新的数据对象，它是某种更通用的数据对象的特殊版本或限制版本。例如，本章中的堆栈数据对象就是更通用的线性表对象的限制版本。堆栈数据对象的实例也是线性表数据对象的实例，而且，所有的堆栈操作都可以利用线性表操作来实现。例如，如果把表的左端定义为栈底，右端定义为栈顶，那么堆栈的添加操作等价于在表的右端进行插入操作，删除操作等价于在表的右端进行删除操作。

根据上述观察，我们期望如果能够明确地把堆栈表示成特殊的线性表，那么可以简化 C++ 堆栈类的实现，而且，可以参照线性表的操作来定义堆栈操作。

若类B是另一个类A的限制版本，那么可以从A派生出B。我们称A为基类，B为派生类。从类A派生出的类B继承了基类A的所有成员——共享成员、保护成员和私有成员。类型为B的每个对象都与A所有的数据成员和函数相关联。类B可以采用如下三种基本方式之一来继承类A的成员：共享成员、保护成员和私有成员。比如，对于共享成员方式，可以采用如下语法形式：

```
class B: public A
```

一个类可以从多个类派生而来。若类B从A和C派生而来，并且以共享成员方式继承A的属性，以私有成员方式继承C的属性，相应的语法形式如下：

```
class B: public A, private C
```

在所有继承方式中，基类A的私有成员仍是A的私有成员，类B的成员不能够访问它们。不同的继承方式仅影响对基类的保护成员和共享成员的访问。

当B按共享成员方式从A派生而来时，A的保护成员成为B的保护成员，A的共享成员成为B的共享成员。所以在编写B的成员代码时，可以直接访问A的共享成员和保护成员，而不能访问A的私有成员。如果X的类型为B，那么，仅当F是B或A的共享成员时，用户才可以使用语句X.F( )来执行X中的F函数。

如果继承方式为保护成员，那么A中的共享成员和保护成员均成为B的保护成员。如果X和F如上所述，那么仅当函数F是B的共享成员时，用户才可以执行X.F()操作。

如果继承方式为私有成员，那么A中的共享成员和保护成员均成为B的私有成员。

所有的继承方式都可以加上一个前缀 virtual，第12章将介绍这个前缀的含义。

### 5.3 公式化描述

由于堆栈是一个受限的线性表（插入和删除操作仅能在表的同一端进行），因此可以参考3.3节的线性表描述，令栈顶元素存储在 element[length-1] 中，栈底元素存储在 element[0] 中。程序5-1中定义的 Stack 类是从程序3-1的 LinearList 类派生而来。由于继承方式为私有成员，因此 LinearList 的共享成员和保护成员都可以被 Stack 的类成员访问，而 LinearList 的私有成员不可以被 Stack 的类成员访问。

程序5-1 公式化描述的堆栈类

```
template<class T>
class Stack :: private LinearList <T>{
// LIFO 对象
public:
    Stack(int MaxStackSize = 10)
        : LinearList<T> (MaxStackSize) {}
    bool IsEmpty() const
        {return LinearList<T>::IsEmpty();}
    bool IsFull() const
        {return (Length() == GetMaxSize());}
    T Top() const
        {if (IsEmpty()) throw OutOfBounds();
         T x; Find(Length(), x); return x;}
    Stack<T>& Add(const T& x)
        {Insert(Length(), x); return *this;}
    Stack<T>& Delete(T& x)
        {LinearList<T>::Delete(Length(), x);
         return *this;}
};
```

Stack 的构造函数简单地调用线性表的构造函数，提供的参数为堆栈的大小 MaxStackSize。没有为 Stack 类定义析构函数，因此当 Stack 类型的对象被删除时，将自动调用 LinearList 的析构函数。IsEmpty() 函数是对线性表相应函数的简单调用。使用操作符 :: 来区分基类和派生类中的同名成员。

在实现函数 IsFull 时，由于 Stack 的成员不能访问 LinearList 的私有成员，因此有一定的困难。为了实现这个函数，必须知道 LinearList<T>::MaxSize 的值。可以通过把 MaxSize 定义成 LinearList 的保护成员来克服这个问题。然而，假如我们在稍后又修改了 LinearList 的定义并删除了成员 MaxSize，那么就不得不同时修改 Stack 的定义。一个比较好的解决方法是为类 LinearList 增加一个保护成员 GetMaxSize()，语法如下：

```
protected:
    int GetMaxSize() const {return MaxSize;}
```

这个函数可以被 Stack 的成员所访问。如果 LinearList 的实现发生变化，那么只需修改 GetMaxSize 的实现，而不必修改它的任何派生类。另一种解决办法是为类 LinearList 定义一个 IsFull 成员函数。

在程序5-1的 Stack<T>::IsFull 代码中，没有使用语法 LinearList<T>::Length()，因为在 Stack



中没有定义Length()函数。

函数Top首先判断堆栈是否为空，如果为空，则不存在栈顶元素，引发一个 OutOfBounds 异常。当堆栈不为空时，调用线性表的Find函数来查找最右端的元素。

函数Add在栈顶添加一个元素x，做法是在线性表的最右端插入元素x。Delete函数删除栈顶元素，并把该元素赋予x。堆栈的删除操作是通过在线性表的最右端执行删除操作而完成的。Add函数和Delete函数均返回变化后的堆栈。Add函数和Delete函数都不能捕获可能由Insert或LinearList::Delete引发的异常。它们把异常留给调用它们的函数来处理。

假定X是一个类型为Stack的对象，当F是Stack的任一个共享函数时，X的使用者可以执行语句X.F。然而，对于LinearList中的函数（如Length），不可以使用语句X.F，因为LinearList中的函数都不是Stack的共享成员（因继承方式为私有成员）。

### 5.3.1 Stack的效率

当T是一个内部数据类型时，堆栈的构造函数和析构函数的复杂性均为  $\Theta(1)$ ，当T是用户自定义的类时，构造函数和析构函数的复杂性均为  $O(\text{MaxStackSize})$ 。其余每个堆栈操作的复杂性均为  $\Theta(1)$ 。注意通过从LinearList派生Stack，一方面大大减少了编码量，另一方面也使程序的可靠性得到很大提高，因为LinearList经过测试并被认为是正确的。然而，不幸的是，代码编写的简化带来了运行效率的损失。例如，为了向堆栈中添加一个元素，首先要确定堆栈的长度length()，然后调用函数Insert()。Insert函数首先必须判断插入操作是否会越界，然后需要付出一个for循环的开销来执行0个元素的移动。为了消除额外的开销，可以把Stack定义成一个基类，而不是派生类。

另一种潜在的问题是派生类Stack也会受到LinearList本身所受限制的影响。例如，必须为数据类型为T的成员定义操作符<<和==，因为前者用于对线性表操作<<的重载，后者用于对LinearList::Search的重载。

### 5.3.2 自定义Stack

程序5-2把Stack类定义为一个基类，所采用的描述形式与线性表相同。堆栈元素存储在数组stack之中，top用于指向栈顶元素。堆栈的容量为MaxTop+1。

程序5-2 自定义Stack

```
template<class T>
class Stack{
// LIFO 对象
public:
    Stack(int MaxStackSize = 10);
    ~Stack () {delete [] stack;}
    bool IsEmpty() const {return top == -1;}
    bool IsFull() const {return top == MaxTop;}
    T Top() const;
    Stack<T>& Add(const T& x);
    Stack<T>& Delete(T& x);
private:
    int top; // 栈顶
    int MaxTop; // 最大的栈顶值
```

```
T *stack; // 堆栈元素数组
};
template<class T>
Stack<T>::Stack(int MaxStackSize)
{// Stack 类构造函数
    MaxTop = MaxStackSize - 1;
    stack = new T[MaxStackSize];
    top = -1;
}
template<class T>
T Stack<T>::Top() const
{// 返回栈顶元素
    if (IsEmpty()) throw OutOfBounds();
    else return stack[top];
}
template<class T>
Stack<T>& Stack<T>::Add(const T& x)
{//添加元素x
    if (IsFull()) throw NoMem();
    stack[++top] = x;
    return *this;
}
template<class T>
Stack<T>& Stack<T>::Delete(T& x)
{// 删除栈顶元素，并将其送入 x
    if (IsEmpty()) throw OutOfBounds();
    x = stack[top--];
    return *this;
}
```

采用一个for循环执行100 000次堆栈添加操作和删除操作来进行实际的运行测试，结果发现程序5-1比程序5-2多用了50%多的时间。

## 练习

1. 对ADT堆栈进行扩充，增加以下函数：

- 1) 确定堆栈的大小（即堆栈中元素的数目）。
- 2) 输入一个堆栈。
- 3) 输出一个堆栈。

扩充基于公式化描述的堆栈定义，增加上述成员函数。编写相应的代码并进行测试。

2. 对ADT堆栈进行扩充，增加以下函数：

1) 把堆栈拆分为两个部分，第一部分包含从栈底开始的一半元素，第二部分包含其余的元素。

2) 合并两个堆栈，把第二个堆栈的所有元素放到第一个堆栈的顶部，并且第二个堆栈中元素的相对次序不发生变化。合并完成后，第二个堆栈为空。

扩充基于公式化描述的堆栈定义，增加上述成员函数。编写相应的代码并进行测试。

3. 基于公式化描述的堆栈定义所存在的缺陷是在创建堆栈时，必须指定MaxStackSize的值。

克服这种缺陷的一种方法是在创建堆栈时取  $\text{MaxTop}=0$ 。如果在 Add 操作期间没有可用的空间来容纳新的元素，可将  $\text{MaxTop}$  变成  $2*\text{MaxTop}+1$ ，然后分配一个新的大小为  $\text{MaxTop}+1$  的数组，并将原数组中的元素复制到新数组中，最后删除原数组。类似地，在删除操作期间，如果表的大小变成数组容量的  $1/4$ ，则可以分配一个容量为原数组一半的新数组，并将原数组中的元素复制到新数组中，最后删除原数组。

1) 采用上述思想重新实现自定义 Stack。构造函数不带参数，其任务是置  $\text{MaxTop}=0$ ，分配一个容量为 1 的数组，并置  $\text{top}$  为 -1。

2) 考察从一个空堆栈开始，连续执行  $n$  个添加和删除操作。假定采用原来的方法时总的执行步数为  $f(n)$ 。证明对于新的方法，执行步数最多为  $cf(n)$ ，其中  $c$  为常量。

## 5.4 链表描述

虽然上一节给出的用数组实现堆栈的方法既优雅又高效，但若同时使用多个堆栈，这种方法将浪费大量的空间。其原因与 3.3.4 节所分析的线性表（用单个数组实现）空间利用率不高的原因相同。不过，若仅同时使用两个堆栈，则是一种例外。可以让一个堆栈从数组的起始位置向后延伸，而让另一个堆栈从数组的结束位置（ $\text{MaxSize}-1$ ）向前延伸，这样可以保持空间的利用率和运行效率。两个堆栈都向数组的中部延伸（如图 5-2 所示）。在描述两个以上的堆栈时，可以借鉴在一个数组中描述多个线性表的方法。不过，这样做在提高空间利用率的同时，却使 Add 操作在最坏情况下的时间复杂性从  $\Theta(1)$  变成了  $O(\text{ArraySize})$ 。Delete 操作的时间复杂性仍然保持为  $\Theta(1)$ 。

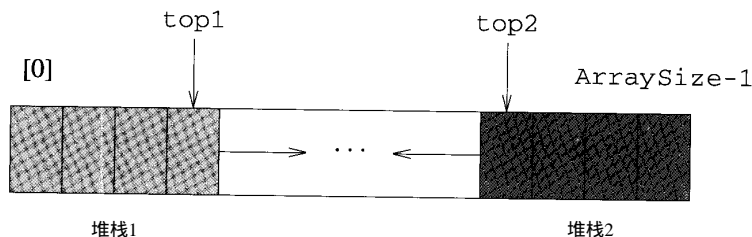


图5-2 一个数组中的两个堆栈

可以采用一个堆栈对应一个链表的方法来高效地描述多个堆栈。这种描述方法使每个堆栈元素多用了—个额外的指针空间。不过，每个堆栈操作的时间复杂性均变为  $\Theta(1)$ 。

在使用链表来表示堆栈时，必须确定链表的哪一端对应于栈顶。如果把链表的右端作为栈顶，那么可以利用链表操作  $\text{Insert}(n,x)$  和  $\text{Delete}(n,x)$  来实现堆栈的插入和删除操作，其中  $n$  为链表中的节点数目。这两种链表操作的时间复杂性均为  $\Theta(n)$ 。另一方面，如果把链表的左端作为栈顶，则可以利用链表操作  $\text{Insert}(0,x)$  和  $\text{Delete}(1,x)$  来实现堆栈的插入和删除操作。这两种链表操作的时间复杂性均为  $\Theta(1)$ 。以上分析表明，应该把链表的左端作为栈顶。

程序 5-3 给出了从 Chain 类派生而来的链表形式的堆栈。链表的左端为栈顶，右端为栈底。

程序 5-3 从 Chain 派生的链表形式的堆栈

```
template<class T>
class LinkedStack : private Chain<T> {
public:
```

```

bool IsEmpty() const
{
    return Chain<T>::IsEmpty();
}
bool IsFull() const;
T Top() const
{
    if (IsEmpty()) throw OutOfBounds();
    T x; Find(1, x); return x;
}
LinkStack<T>& Add(const T& x)
{
    Insert(0, x); return *this;
}
LinkStack<T>& Delete(T& x)
{
    Chain<T>::Delete(1, x); return *this;
}
};
template<class T>
bool LinkStack<T>::IsFull() const
{
    // 堆栈是否满?
    try {ChainNode<T> *p = new ChainNode<T>;
        delete p; return false;}
    catch (NoMem) {return true;}
}

```

IsFull的实现不够优雅，因为它是通过实际创建一个类型为 Node的节点来判断能否添加一个新元素。判断之后必须删除所创建的节点，因为并不打算使用所创建的节点。

与程序 5-1 的情形一样，为了提高运行效率，也可以自定义一个用链表形式的堆栈。相应的代码见程序 5-4。

对于执行 10 000 次添加和删除操作的 for 循环来说，程序 5-3 的代码将比程序 5-4 的代码多耗时 25%。

程序 5-4 自定义链表形式的堆栈

```

template <class T>
class Node{
    friend LinkStack<T>;
private:
    T data;
    Node<T> *link;
};
template<class T>
class LinkStack {
public:
    LinkStack () {top = 0;}
    ~LinkStack();
    bool IsEmpty() const {return top==0;}
    bool IsFull() const;
    T Top() const;
    LinkStack<T>& Add(const T& x);
    LinkStack<T>& Delete(T& x);
private:
    Node<T> *top; // 指向栈顶节点
};
template<class T>
LinkStack<T>::~LinkStack()

```

```
// 析构函数
Node<T> *next;
while (top) {
    next = top->link;
    delete top;
    top = next;
}
}
template<class T>
bool LinkedStack<T>::IsFull() const
// 堆栈是否满?
try {Node<T> *p = new Node<T>;
    delete p;
    return false;}
catch (NoMem) {return true;}
}
template<class T>
T LinkedStack<T>::Top() const
// 返回栈顶元素
if (IsEmpty()) throw OutOfBounds();
return top->data;
}
template<class T>
LinkedStack<T>& LinkedStack<T>::Add(const T& x)
// 添加元素 x
Node<T> *p = new Node<T>;
p->data = x;
p->link = top;
top = p;
return *this;
}
template<class T>
LinkedStack<T>& LinkedStack<T>::Delete(T& x)
// 删除栈顶元素, 并将其送入 x
if (IsEmpty()) throw OutOfBounds();
x = top->data;
Node<T> *p = top;
top = top->link;
delete p;
return *this;
}
```

## 练习

4. 1) 编写一个测试程序, 用它来测量 100 000 个交替的堆栈添加和删除操作所需要的运行时间。分别对程序 5-1、5-2、5-3、5-4 以及练习 3 进行测试。

2) 交替的添加和删除操作对于练习 3 的代码来说是最理想的情形。请给出能使练习 3 得到最坏的时间复杂性的测试数据, 同时利用该数据测量 100 000 次堆栈操作所需要的时间。

5. 扩充 LinkedStack 类的定义, 增加以下堆栈操作:

1) 确定堆栈的大小 (即堆栈中元素的数目)。

2) 输入一个堆栈。

3) 输出一个堆栈。

6. 扩充LinkedStack类的定义, 增加以下操作:

1) 把堆栈拆分为两个部分, 第一部分包含从栈底开始的一半元素, 第二部分包含其余的元素。

2) 合并两个堆栈, 把第二个堆栈的所有元素放到第一个堆栈的顶部, 并且第二个堆栈中元素的相对次序不发生变化。合并完成后, 第二个堆栈为空。

## 5.5 应用

### 5.5.1 括号匹配

在这个问题中将要匹配一个字符串中的左、右括号。例如, 字符串  $a*(b+c)+d$  在位置1和4有左括号, 在位置8和11有右括号。位置1的左括号匹配位置11的右括号, 位置4的左括号匹配位置8的右括号。对于字符串  $(a+b)($ , 位置6的右括号没有可匹配的左括号, 位置7的左括号没有可匹配的右括号。我们的目标是编写一个 C++ 程序, 其输入为一个字符串, 输出为相互匹配的括号以及未能匹配的括号。注意, 括号匹配问题可用来解决 C++ 程序中的 { 和 } 的匹配问题。

可以观察到, 如果从左至右扫描一个字符串, 那么每个右括号将与最近遇到的那个未匹配的左括号相匹配。这种观察结果使我们联想到可以在从左至右的扫描过程中把所遇到的左括号存放到堆栈内。每当遇到一个右括号时, 就将其与栈顶的左括号 (如果存在) 相匹配, 同时从栈顶删除该左括号。程序 5-5 给出了完整的 C++ 程序。图 5-3 给出了某次运行所得到的输入/输出结果。程序 5-5 的时间复杂性为  $\Theta(n)$ , 其中  $n$  为输入串的长度。

程序5-5 产生匹配括号的程序

```
#include <iostream.h>
#include <string.h>
#include <stdio.h>
#include "stack.h"

const int MaxLength = 100; // 最大的字符串长度
void PrintMatchedPairs(char *expr)
{
    // 括号匹配
    Stack<int> s(MaxLength);
    int j, length = strlen(expr);
    // 从表达式 expr 中搜索 ( 和 )
    for (int i = 1; i <= length; i++) {
        if (expr[i - 1] == ' ( ' ) s.Add(i);
        else if (expr[i - 1] == ' ) ' )
            try{s.Delete(j);
                cout << j << ' ' << i << endl;}
            catch (OutOfBounds)
                { cout << "No match for right parenthesis" << " at " << i << endl;}
    }
    // 堆栈中所剩下的(都是未匹配的
    while(!s.IsEmpty()) {
```

```
s.Delete(j);
cout << "No match for left parenthesis at " << j << endl;
}
void main(void)
{
    char expr[MaxLength];
    cout << "Type an expression of length at most " << MaxLength << endl;
    cin.getline(expr, MaxLength);
    cout << "The pairs of matching parentheses in" << endl;
    puts (expr);
    cout << "are" << endl;
    PrintMatchnedPairs(expr);
}
```

```
Type an expression of length at most 100
(d+(a+b)*c*(d+e)-f))((
The pairs of matching parentheses in the expression
(d+(a+b)*c*(d+e)-f))((
are
4 8
12 16
1 19
No match for right parenthesis at 20
22 23
No match for left parenthesis at 21
```

图5-3 括号匹配程序运行示例

### 5.5.2 汉诺塔

汉诺塔 (Towers of Hanoi) 问题来自一个古老的传说：在世界刚被创建的时候有一座钻石宝塔 (塔1)，其上有64个金碟 (如图5-4所示)。所有碟子按从大到小的次序从塔底堆放至塔顶。紧挨着这座塔有另外两个钻石宝塔 (塔2和塔3)。从世界创始之日起，婆罗门的牧师们就一直在试图把塔1上的碟子移动到塔2上去，其间借助于塔3的帮助。由于碟子非常重，因此，每次只能移动一个碟子。另外，任何时候都不能把一个碟子放在比它小的碟子上面。按照这个传说，当牧师们完成他们的任务之后，世界末日也就到了。

在汉诺塔问题中，已知  $n$  个碟子和3座塔。初始时所有的碟子按从大到小次序从塔1的底部堆放至顶部，我们需要把碟子都移动到塔2，每次移动一个碟子，而且任何时候都不能把大碟子放到小碟子的上面。在继续往下阅读之前，可以先尝试对  $n=2,3$  和4来解决这个问题。

一个非常优雅的解决办法是使用递归。为了把最大的碟子移动到塔2，必须把其余  $n-1$  个碟子移动到塔3，然后把最大的碟子移动到塔2。接下来是把塔3上的  $n-1$  个碟子移动到塔2，为此可以利用塔2和塔1。可以完全忽视塔2上已经有一个碟子的事实，因为这个碟子比塔3上将要移过来的任一个碟子都大，因此，可以在它上面堆放任何碟子。程序5-6给出了按递归方式实现的C++代码。初始调用的语句是 TowersOfHanoi( $n,1,2,3$ )。程序5-6的正确性很容易证明。



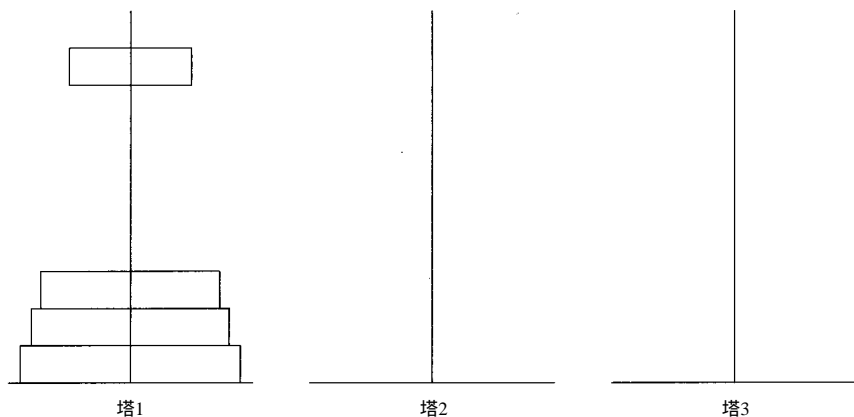


图5-4 汉诺塔

程序5-6 求解汉诺塔问题的递归程序

```

void TowersOfHanoi(int n, int x, int y, int z)
{
    //把 n 个碟子从塔x 移动到塔 y, 可借助于塔 z
    if (n > 0) {
        TowersOfHanoi(n-1, x, z, y);
        cout << "Move top disk from tower " << x << " to top of tower " << y << endl;
        TowersOfHanoi(n-1, z, y, x);
    }
}

```

程序5-6所花费的时间正比于所输出的信息行数，而信息行的数目则等价于碟子移动的次數。考察程序5-6，可以得到碟子的移动次數  $moves(n)$  如下：

$$moves(n) = \begin{cases} 0 & n=0 \\ 2moves(n-1) + 1 & n>0 \end{cases}$$

可以使用第2章介绍的叠代方法（见程序2-20）来计算这个公式。得到的结果应为  $moves(n)=2^n-1$ 。可以证明， $2^n-1$  实际上是最少的移动次数。在婆罗门宝塔中  $n=64$ ，因此婆罗门牧师们用不了多少年就可以完成任务。根据上面的公式，可以断定函数 TowersOfHanoi 的复杂性为  $\Theta(2^n)$ 。

程序5-6的输出给出了把碟子从塔1移动到塔2所需要的碟子移动次序。假定希望给出每次移动之后三座塔的状态（即塔上的碟子及其次序），那么必须在内存中保留塔的状态，并在每次移动碟子之后，对塔的状态进行修改。这样每移动一个碟子时，就可以在一个输出设备（如计算机屏幕、打印机等）上输出塔的信息。

由于从每个塔上移走碟子时是按照 LIFO 的方式进行的，因此可以把每个塔表示成一个堆栈。三座塔在任何时候都总共拥有  $n$  个碟子，因此，如果使用链表形式的堆栈，只需申请  $n$  个元素所需要的空间。如果使用的是基于公式化描述的堆栈，塔1和塔2的容量都必须是  $n$ ，而塔3的容量必须为  $n-1$ ，因而所需要的空间总数为  $3n-1$ 。前面的分析已经指出，汉诺塔问题的复杂性是以  $n$  为指数的函数，因此在可以接受的时间范围内，只能解决  $n$  值比较小（如  $n=30$ ）的汉诺塔问题。对于这些较小的  $n$  值，基于公式描述和基于链表描述的堆栈在空间需求上的差别相当小，因此可以随意使用。

程序 5-7 的代码使用了基于公式描述的堆栈。TowersOfHanoi(n) 是递归函数 Hanoi::TowersOfHanoi 的预处理程序，它是根据程序 5-6 的模式来设计的。预处理程序创建三个堆栈 S[1:3] 用来存储 3 座塔的状态。所有的碟子从 1（最小碟子）到 n（最大碟子）编号，因此每个堆栈的类型均为 int。如果没有足够的空间来创建三个堆栈，堆栈构造函数将引发一个类型为 NoMem 的异常，预处理程序终止执行。如果有足够的空间，预处理程序将调用 Hanoi::TowersOfHanoi。在该程序中没有给出由 Hanoi::TowersOfHanoi 所调用的函数 ShowState，原因是该函数的实现取决于输出设备的性质（如计算机屏幕、打印机等）。

程序 5-7 使用堆栈求解汉诺塔问题

```
class Hanoi{
    friend void TowersOfHanoi(int);
public:
    void TowersOfHanoi(int n, int x, int y, int z);
private:
    Stack<int> *S[4];
};

void Hanoi::TowersOfHanoi(int n, int x, int y, int z)
{
    // 把 n 个碟子从塔 x 移动到塔 y，可借助于塔 z
    int d; // 碟子编号
    if (n > 0) {
        TowersOfHanoi(n-1, x, z, y);
        S[x]->Delete(d); // 从 x 中移走一个碟子
        S[y]->Add(d); // 把这个碟子放到 y 上
        ShowState();
        TowersOfHanoi(n-1, z, y, x);
    }
}

void TowersOfHanoi(int n)
{
    // Hanoi::towersOfHanoi 的预处理程序
    Hanoi X;
    X.S[1] = new Stack<int> (n);
    X.S[2] = new Stack<int> (n);
    X.S[3] = new Stack<int> (n);

    for (int d = n; d > 0; d--) // 初始化
        X.S[1]->Add(d); // 把碟子 d 放到塔 1 上

    // 把塔 1 上的 n 个碟子移动到塔 3 上，借助于塔 2 的帮助
    X.TowersOfHanoi(n, 1, 2, 3);
}
```

### 5.5.3 火车车厢重排

一列货运列车共有  $n$  节车厢，每节车厢将停放在不同的车站。假定  $n$  个车站的编号分别为  $1 \sim n$ ，货运列车按照第  $n$  站至第 1 站的次序经过这些车站。车厢的编号与它们的目的地相同。为了便于从列车上卸掉相应的车厢，必须重新排列车厢，使各车厢从前至后按编号  $1$  到  $n$  的次序排列。当所有的车厢都按照这种次序排列时，在每个车站只需卸掉最后一节车厢即可。我们在一

个转轨站里完成车厢的重排工作，在转轨站中有一个入轨、一个出轨和  $k$  个缓冲铁轨（位于入轨和出轨之间）。图5-5a 给出了一个转轨站，其中有  $k=3$  个缓冲铁轨  $H1$ 、 $H2$  和  $H3$ 。开始时， $n$  节车厢的货车从入轨处进入转轨站，转轨结束时各车厢从右到左按照编号 1 至编号  $n$  的次序离开转轨站（通过出轨处）。在图5-5a 中， $n=9$ ，车厢从后至前的初始次序为 5, 8, 1, 7, 4, 2, 9, 6, 3。图5-5b 给出了按所要求的次序重新排列后的结果。

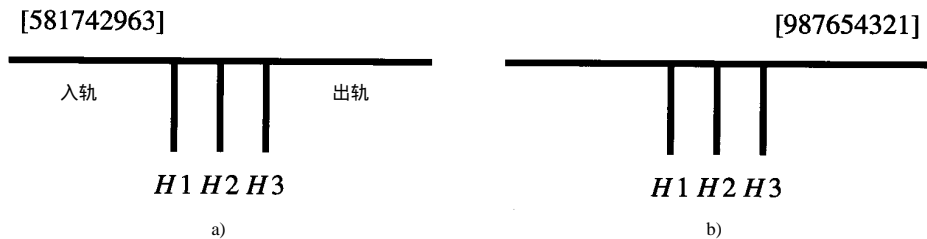


图5-5 具有三个缓冲铁轨的转轨站

a) 开始 b) 结束

为了重排车厢，需从前至后依次检查入轨上的所有车厢。如果正在检查的车厢就是下一个满足排列要求的车厢，可以直接把它放到出轨上去。如果不是，则把它移动到缓冲铁轨上，直到按输出次序要求轮到它时才将它放到出轨上。缓冲铁轨是按照 LIFO 的方式使用的，因为车厢的进和出都是在缓冲铁轨的顶部进行的。在重排车厢过程中，仅允许以下移动：

- 车厢可以从入轨的前部（即右端）移动到一个缓冲铁轨的顶部或出轨的左端。
- 车厢可以从一个缓冲铁轨的顶部移动到出轨的左端。

考察图5-5a。3号车厢在入轨的前部，但由于它必须位于 1 号和 2 号车厢的后面，因此不能立即输出 3 号车厢，可把 3 号车厢送入缓冲铁轨  $H1$ 。下一节车厢是 6 号车厢，也必须送入缓冲铁轨。如果把 6 号铁轨送入  $H1$ ，那么重排过程将无法完成，因为 3 号车厢位于 6 号车厢的后面，而按照重排的要求，3 号车厢必须在 6 号车厢之前输出。因此可把 6 号车厢送入  $H2$ 。下一节车厢是 9 号车厢，被送入  $H3$ ，因为如果把它送入  $H1$  或  $H2$ ，重排过程也将无法完成。请注意：当缓冲铁轨上的车厢编号不是按照从顶到底的递增次序排列时，重排任务将无法完成。至此，缓冲铁轨的当前状态如图5-6a 所示。

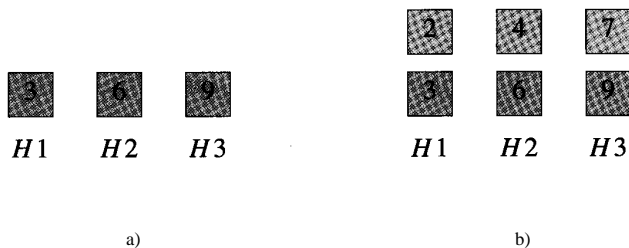


图5-6 缓冲铁轨中间状态

接下来处理 2 号车厢，它可以被送入任一个缓冲铁轨，因为它能满足缓冲铁轨上车厢编号必须递增排列的要求，不过，应优先把 2 号车厢送入  $H1$ ，因为如果把它送入  $H3$ ，将没有空间来移动 7 号车厢和 8 号车厢。如果把 2 号车厢送入  $H2$ ，那么接下来的 4 号车厢必须被送入  $H3$ ，这样将无法移动后面的 5 号、7 号和 8 号车厢。新的车厢  $u$  应送入这样的缓冲铁轨：其顶部的车厢编号

$v$  满足  $v > u$ ，且  $v$  是所有满足这种条件的缓冲铁轨顶部车厢编号中最小的一个编号。只有这样才能使后续的车厢重排所受到的限制最小。我们将使用这条分配规则（assignment rule）来选择缓冲铁轨。

接下来处理4号车厢时，三个缓冲铁轨顶部的车厢分别是2号、6号和9号车厢。根据分配规则，4号车厢应送入  $H_2$ 。这之后，7号车厢被送入  $H_3$ 。图5-6b 给出了当前的状态。接下来，1号车厢被送至出轨，这时，可以把  $H_1$  中的2号车厢送至出轨。之后，从  $H_1$  输出3号车厢，从  $H_2$  输出4号车厢。至此，没有可以立即输出的车厢了。

接下来的8号车厢被送入  $H_1$ ，然后5号车厢从入轨处直接输出到出轨处。这之后，从  $H_2$  输出6号车厢，从  $H_3$  输出7号车厢，从  $H_1$  输出8号车厢，最后从  $H_3$  输出9号车厢。

对于图5-5a 的初始排列次序，在进行车厢重排时，只需三个缓冲铁轨就够了，而对于其他的初始次序，可能需要更多的缓冲铁轨。例如，若初始排列次序为  $1, n, n-1, \dots, 2$ ，则需要  $n-1$  个缓冲铁轨。

为了实现上述思想，用  $k$  个链表形式的堆栈来描述  $k$  个缓冲铁轨。之所以采用链表形式的堆栈而不是公式化形式的堆栈，原因在于前者仅需要  $n-1$  个元素。函数 Railroad（见程序 5-8）用于确定重排  $n$  个车厢，它最多可使用  $k$  个缓冲铁轨并假定车厢的初始次序为  $p[1:n]$ 。如果不能成功地重排，Railroad 返回 false，否则返回 true。如果由于内存不足而使函数失败，则引发一个异常 NoMem。

函数 Railroad 在开始时创建一个指向堆栈的数组  $H$ ， $H[i]$  代表缓冲铁轨  $i$ ， $1 \leq i \leq k$ 。NowOut 是下一个欲输出至出轨的车厢号。minH 是各缓冲铁轨中最小的车厢号，minS 是 minH 号车厢所在的缓冲铁轨。

在 for 循环的第  $i$  次循环中，首先从入轨处取车厢  $p[i]$ ，若  $p[i] = \text{NowOut}$ ，则将其直接送至出轨，并将 NowOut 的值增 1，这时，有可能会从缓冲铁轨中输出若干节车厢（通过 while 循环把它们送至出轨处）。如果  $p[i]$  不能直接输出，则没有车厢可以被输出，按照前述的铁轨分配规则把  $p[i]$  送入相应的缓冲铁轨之中。

程序 5-8 火车车厢重排程序

```
bool Railroad(int p[], int n, int k)
// k 个缓冲铁轨，车厢初始排序为 p[1:n]
// 如果重排成功，返回 true，否则返回 false
// 如果内存不足，则引发异常 NoMem。
// 创建与缓冲铁轨对应的堆栈
LinkedList<int> *H;
H = new LinkedList<int> [k + 1];
int NowOut = 1; // 下一次要输出的车厢
int minH = n+1; // 缓冲铁轨中编号最小的车厢
int minS; // minH 号车厢对应的缓冲铁轨
// 车厢重排
for (int i = 1; i <= n; i++)
    if (p[i] == NowOut) { // 直接输出 t
        cout << "Move car " << p[i] << " from input to output" << endl;
        NowOut++;
    }
    // 从缓冲铁轨中输出
    while (minH == NowOut) {
        Output(minH, minS, H, k, n);
```

```

        NowOut++;
    }
}
else { // 将 p[i] 送入某个缓冲铁轨
    if (!Hold(p[i], minH, minS, H, k, n))
        return false;
    return true;
}
}

```

程序5-9和5-10分别给出了Railroad中所使用的函数Output和Hold。Output用于把一节车厢从缓冲铁轨送至出轨处，它同时将修改minS和minH。函数Hold根据车厢分配规则把车厢c送入某个缓冲铁轨，必要时，它也需要修改minS和minH。

程序5-9 程序5-8中所使用的Output 函数

```

void Output(int& minH, int& minS, LinkedStack<int> H[], int k, int n)
{ //把车厢从缓冲铁轨送至出轨处，同时修改 minS和minH
    int c; // 车厢索引
    // 从堆栈minS中删除编号最小的车厢 minH
    H[minS].Delete(c);
    cout << "Move car " << minH << " from holding track " << minS << " to output" << endl;
    // 通过检查所有的栈顶，搜索新的 minH和minS
    minH = n + 2;
    for (int i = 1; i <= k; i++)
        if (!H[i].IsEmpty() && (c = H[i].Top()) < minH) {
            minH = c;
            minS = i;
        }
}

```

程序5-10 程序5-8中所使用的Hold函数

```

bool Hold(int c, int& minH, int &minS, LinkedStack<int> H[], int k, int n)
{ // 在一个缓冲铁轨中放入车厢 c
    // 如果没有可用的缓冲铁轨，则返回 false
    // 如果空间不足，则引发异常 NoMem
    // 否则返回true
    // 为车厢c寻找最优的缓冲铁轨
    // 初始化
    int BestTrack = 0, // 目前最优的铁轨
    BestTop = n + 1, // 最优铁轨上的头辆车厢
    x; // 车厢索引
    //扫描缓冲铁轨
    for (int i = 1; i <= k; i++)
        if (!H[i].IsEmpty()) { // 铁轨 i 不空
            x = H[i].Top();
            if (c < x && x < BestTop) {
                //铁轨 i 顶部的车厢编号最小
                BestTop = x;
                BestTrack = i;
            }
        }
}

```

```

else // 铁轨 i 为空
    if (!BestTrack) BestTrack = i;
if (!BestTrack) return false; //没有可用的铁轨
//把车厢c 送入缓冲铁轨
H[BestTrack].Add(c);
cout << "Move car " << c << " from input " << BestTrack << endl;
//必要时修改 minH 和 minS
if (c < minH) {minH = c; minS = BestTrack;}
return true;
}

```

为了计算程序 5-8 的时间复杂性，我们首先注意到 Output 和 Hold 的复杂性均为  $\Theta(k)$ 。由于在 Railroad 的 while 循环中最多可以输出  $n-1$  节车厢，且在 else 语句中最多有  $n-1$  节车厢被送入缓冲铁轨，因此，由函数 Output 和 Hold 所消耗的总时间为  $O(kn)$ 。Railroad 中 for 循环的其余部分需耗时  $\Theta(n)$ 。所以，程序 5-8 总的时间复杂性为  $O(kn)$ 。若使用一个平衡折半搜索树（如 AVL 树）来存储缓冲铁轨顶部的车厢编号（见第 11 章），则程序的复杂性可以降至  $O(n \log k)$ 。在使用平衡折半搜索树时，可以重写函数 Output 和 Hold，以使其具有复杂性  $O(\log k)$ 。对于本应用，仅当  $k$  很大时，才推荐使用平衡折半搜索树。

#### 5.5.4 开关盒布线

开关盒布线问题是这样的：给定一个矩形布线区域，其外围有若干针脚。两个针脚之间通过布设一条金属线路而实现互连。这条线路被称为电线，被限制在矩形区域内。如果两条电线发生交叉，则会发生电流短路。所以，不允许电线间的交叉。每对互连的针脚被称为网组。我们的目标是确定对于给定的网组，能否合理地布设电线以使其不发生交叉。图 5-7a 给出了一个布线的例子，其中有八个针脚和四个网组。四个网组分别是 (1,4)，(2,3)，(5,6) 和 (7,8)。图 5-7b 给出的布线方案有交叉现象发生（(1,4) 和 (2,3) 之间），而图 5-7c 则没有交叉现象发生。由于四个网组可以通过合理安排而不发生交叉，因此可称其为可布线开关盒（routable switch box）。（在具体实现时，还需要在两个相邻的电线间留出一定的间隔，为使问题简化，本应用中忽略这个额外的要求）。我们要解决的问题是，给定一个开关盒布线实例，确定它是不是一个可布线的。

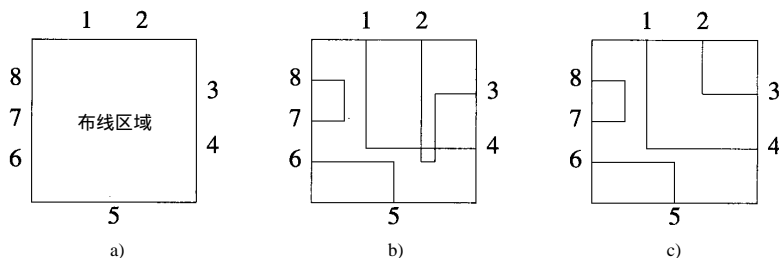


图5-7 开关盒布线示例

图 5-7b 和 5-7c 中的电线都是由平行于  $x$  轴和  $y$  轴的垂直线段构成的，当然也可以使用不与  $x$  轴和  $y$  轴平行的线段。

为了解决开关盒布线问题，我们注意到，当两个针脚互连时，其电线把布线区分成两个分

区。例如，当(1,4)互连时，就得到了两个分区，一个分区包含针脚2和3，另一个分区包含针脚5~8。现在如果有一个网组，其两个针脚分别位于这两个不同的分区，那么这个网组是不可以布线的，因而整个电路也是不可布线的。如果没有这样的网组，则可以继续判断每个独立的分区是不是可布线的。为此，可以从一个分区中取出一个网组，利用该网组把这个分区又分成两个子分区，如果任一个网组的两个针脚都分布在同一个子分区之中（即不会出现两个针脚分别位于两个子分区的情形），那么这个分区就是可布线的。

为了实现上述策略，可以按顺时针或反时针方向沿着开关盒的外围进行遍历，可从任意一个针脚开始。例如，如果按顺时针方向从针脚1开始遍历图5-7a中的针脚，那么将依次检查针脚1, 2, ..., 8。针脚1和4属于同一个网组，那么在针脚1至针脚4之间出现的所有针脚构成了第一个分区，而在针脚4至针脚1之间出现的所有针脚构成了第二个分区。把针脚1放入堆栈，然后继续处理，直至遇到针脚4。这个过程使我们仅在处理完一个分区之后才能进入下一个分区。下一个针脚是针脚2，它与针脚3同属一个网组，它们又把当前分区分成两个子分区。与前面的做法一样，把针脚2放入堆栈，然后继续处理直至遇到针脚3。由于针脚3与针脚2属同一个网组，而针脚2正处在栈顶，这表明已经处理完一个子分区，因此可将针脚2从栈顶删除。接下来将遇到针脚4，由于与之互连的针脚1正处在栈顶，因此当前的分区已经处理完毕，可从栈顶删除针脚1。按照这种方法继续进行下去，直至检查完八个针脚，堆栈变空，所创建的分区都已处理完毕为止。

那么，对于不可布线的开关盒将会出现什么样的情况呢？假定图5-7a中的网组是：(1,5)，(2,3)，(4,7)和(6,8)。初始时，针脚1和2被放入堆栈。在检查针脚3时，将针脚2从栈顶删除。接下来针脚4被放入堆栈，因为针脚4与栈顶的针脚不能构成一个网组。在检查针脚5时，它也被放入堆栈。尽管已经遇到了针脚1和针脚5，但还不能结束由这两个针脚所定义的第一个分区的处理过程，因为针脚4的布线将不得不跨越这个分区的边界。因此，当完成对所有针脚的检查时，堆栈不会变空。

程序5-11给出了按上述策略实现的C++程序。它要求对每个网组进行编号，并且每个针脚也得有一个对应的网组编号。所以，对于图5-7c中的例子，输入数组net=[1,2,2,1,3,3,4,4]。程序5-11的复杂性为 $\Theta(n)$ ，其中n为针脚的数目。

程序5-11 开关盒布线

```
bool CheckBox(int net[ ], int n)
// 确定开关盒是否可布线
Stack<int> *s = new Stack<int> (n);
//顺时针扫描各网组
for (int i = 0; i < n; i++) {
    //检查net[i]
    if (!s->IsEmpty()) {
        if (net[i] == net[s->Top()]) {
            // net[i] 可布线，从堆栈中删除
            int x;
            s->Delete(x);
        }
        else s->Add(i);
    }
    else s->Add(i);
}
// 是否有不可布线的网组？
if (s->IsEmpty()) {
```



```

delete s;
cout << "Switch box is routable" << endl;
return true;}
delete s;
cout << "Switch box is not routable" << endl;
return false;
}

```

### 5.5.5 离线等价类问题

离线等价类问题的定义见3.8.3节。这个问题的输入是元素数目 $n$ 、关系数目 $r$ 以及 $r$ 对关系，问题的目标是把 $n$ 个元素分配至相应的等价类。程序5-12给出了解决离线等价类问题的C++程序。在程序5-12a中，输入为 $n$ 、 $r$ 和 $r$ 对关系，对于每个元素都建立了一个相应的链表。与元素 $i$ 对应的链表 $chain[i]$ 中包含所有这样的元素 $j$ ： $(i, j)$ 或 $(j, i)$ 是所输入的关系。程序5-12b用于输出等价类，其中使用了一个数组 $out$ ，当且仅当 $i$ 已经被作为某个等价类的成员输出时， $out[i]=true$ 。堆栈 $stack$ 用于定位一个等价类中所有的元素。在这个等价类中含有当前等价类中所有的元素。

为了找到下一个等价类中的第一个成员，可以扫描数组 $out$ 以寻找尚未被输出的元素。如果没有这样的元素，则表明不再有新的等价类。如果找到一个这样的元素，则开始搜索下一个等价类。把这个元素放入堆栈，然后依次对堆栈中的元素进行检查，看看这些元素是否与该元素等价。具体的检查方法是：从堆栈中删除一个元素 $m$ ，然后检查 $chain[m]$ 中的所有元素。当堆栈为空时，在当前等价类中将不会存在这样的成员 $m$ ： $(m, p)$ 是所输入的关系，而 $p$ 尚未被输出（因为 $p$ 一定位于 $chain[m]$ 之中，所以从堆栈中删除 $m$ 时， $p$ 肯定已经被输出）。因此，在每次do循环过程中所输出的元素都构成了一个等价类。

为了分析等价类程序的复杂性，可以注意到，由于每次链表的插入操作都是在链表的首部进行的，因此每次插入操作需耗时 $\Theta(1)$ 。程序5-12a需耗时 $\Theta(n+r)$ （用于输入和链表初始化）。对于程序5-12b，可以发现，每个元素都仅被输出一次，每个元素都只进堆栈一次，并被从堆栈中删除一次，因此，执行堆栈添加和删除操作所需要的总时间为 $\Theta(n)$ 。当从堆栈中删除一个元素时，该元素对应链表中的所有元素也将被删除。由于每次删除都是从链表首部进行的，因此每次删除需耗时 $\Theta(1)$ 。在程序5-12a中输入结束时，所有 $n$ 个链表中的元素总数为 $2r$ ，而在程序5-12b中所有 $2r$ 个链表元素都将被删除，因此删除链表元素共需耗时 $\Theta(r)$ 。这样，程序5-12总的的时间复杂性为 $\Theta(n+r)$ 。

程序5-12a 离线等价类程序（一）

```

void main(void)
{// 离线等价类问题
    int n, r;
    //输入n 和r
    cout << "Enter number of elements" << endl;
    cin >> n;
    if (n < 2) {cerr << "Too few elements" << endl; exit(1);}
    cout << "Enter number of relations" << endl;
    cin >> r;
    if (r < 1) {cerr << "Too few relations" << endl; exit(1);}
    //创建一个指向n个链表的数组

```

```

Chain<int> *chain;
try {chain = new Chain<int> [n+1];}
catch (NoMem) {cerr << "Out of memory" << endl; exit(1);}
//输入 r个关系，并存入链表
for (int i = 1; i <= r; i++) {
    cout << "Enter next relation/pair" << endl;
    int a, b;
    cin >> a >> b;
    chain[a].Insert(0,b);
    chain[b].Insert(0,a);
}

```

---

#### 程序5-12b 离线等价类程序 (二)

---

```

//对欲输出的等价类进行初始化
LinkedStack<int> stack;
bool *out;
try {out = new bool [n+1];}
catch (NoMem) {cerr << "Out of memory" << endl; exit(1);}
for (int i = 1; i <= n; i++)
    out[i] = false;
//输出等价类
for (int i = 1; i <= n; i++)
    if (!out[i]) {//开始一个新的等价类
        cout << "Next class is: " << i << ' ';
        out[i] = true;
        stack.Add(i);
        //从堆栈中取其余的元素
        while (!stack.IsEmpty()) {
            int *q, j;
            stack.Delete(j);
            //链表chain[j]中的元素都在同一个等价类中，使用遍历器 c来取这些元素
            ChainIterator<int> c;
            q = c.Initialize(chain[j+1]);
            while (q){
                if (!out[*q]) {
                    cout << "q << ' ';
                    out[*q] = true;
                    stack.Add(*q);}
                q = c.Next();
            }
        }
        cout << endl;
    }
    cout << endl << "End of class list" << endl;
}

```

注意，在程序5-12中并未删除为chain和out所分配的空间。由于在程序结束时，这些空间会被自动释放，因此，没有必要特意去删除它们。如果把程序5-12改编成一个函数，那么必须

增加相应的语句来删除这两个数组所占用的空间，以便这些空间能为程序的其他部分所用。

### 5.5.6 迷宫老鼠

#### 1. 问题描述

迷宫 (maze) 是一个矩形区域，它有一个入口和一个出口。在迷宫的内部包含不能穿越的墙或障碍。在图 5-8 所示的迷宫中，障碍物沿着行和列放置，它们与迷宫的矩形边界平行。迷宫的入口在左上角，出口在右下角。

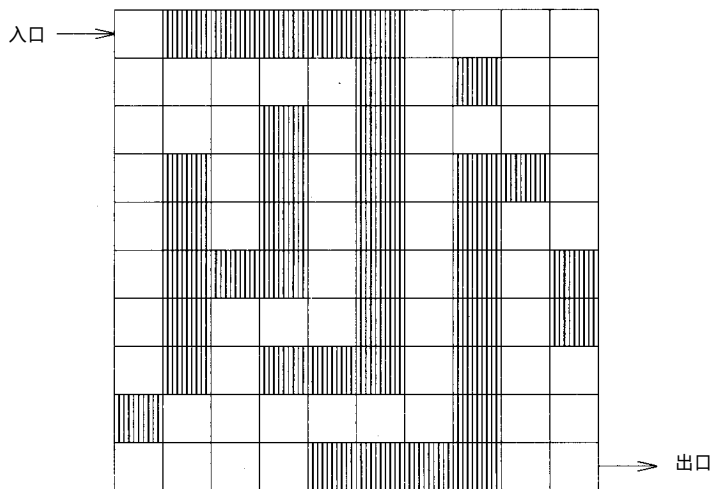


图5-8 迷宫

假定用  $n \times m$  的矩阵来描述迷宫，位置  $(1,1)$  表示入口， $(n,m)$  表示出口， $n$  和  $m$  分别代表迷宫的行数和列数。迷宫中的每个位置都可用其行号和列号来指定。在矩阵中，当且仅当在位置  $(i,j)$  处有一个障碍时其值为 1，否则其值为 0。图 5-9 给出了图 5-8 中迷宫对应的矩阵描述。迷宫老鼠 (rat in a maze) 问题要求寻找一条从入口到出口的路径。路径是由一组位置构成的，每个位置上都没有障碍，且每个位置（第一个除外）都是前一个位置的东、南、西或北的邻居（如图 5-10 所示）。

```

0 1 1 1 1 1 0 0 0 0
0 0 0 0 0 1 0 1 0 0
0 0 0 1 0 1 0 0 0 0
0 1 0 1 0 1 0 1 1 0
0 1 0 1 0 1 0 1 0 0
0 1 1 1 0 1 0 1 0 1
0 1 0 0 0 1 0 1 0 1
0 1 0 1 1 1 0 1 0 0
1 0 0 0 0 0 0 1 0 0
0 0 0 0 1 1 1 1 0 0

```

图5-9 图5-8中迷宫的矩阵描述

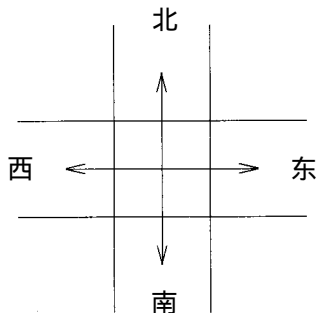


图5-10 迷宫任意位置处的4种移动方向

下面将要编写程序来解决迷宫老鼠问题。假定程序中所使用的迷宫是正方形的（即行数等

于列数)，且迷宫足够小，以便能在目标计算机的内存中描述整个迷宫。所编写的程序应是一个独立的产品，想要在迷宫中寻找路径的人可以直接使用它。

## 2. 设计

可以采用自顶向下的模块化方法来设计这个程序。不难看出，这个程序可以划分为三个部分：输入迷宫、寻找路径和路径输出。每个部分可以专门用一个程序模块来实现。第四个模块用来显示欢迎信息、软件名称及作者信息，这个模块与其他三个模块之间没有任何直接关系，主要用于增强程序界面的友好性。

寻找路径的模块不会直接与用户打交道，因此不必提供帮助机制，也不需要由菜单驱动。其他三个模块都会与用户进行交互，因此应多花一些精力来设计它们的用户接口。好的用户接口能使用户喜欢使用你的程序。

首先来设计欢迎模块。我们希望显示如下信息：

**Welcome To**  
**RAT IN A MAZE**  
© Joe Bloe, 1998

如果觉得按这种形式显示信息显得有些单调，可以引入其他手段以得到满意的效果。比如，可以用多种颜色来显示这些信息，可以使每行文字的大小不一样（甚至每个字符的大小也不一样），也可以让每个字符按一定的时间间隔依次出现。除此以外，还可以考虑使用声音效果。信息显示的时间也需要确定，显示的时间应足够长，以使用户能够读完它，但也不能太长（以至于用户打哈欠）。因此，欢迎信息（乃至整个用户接口）的设计需要一定的艺术技巧。

输入模块应告诉用户需输入一个由0和1组成的矩阵。为此，需要确定是按逐行方式还是逐列的方式来输入矩阵。由于逐行方式是比较自然的方式，因此可要求用户按逐行方式输入矩阵。在输入矩阵数据之前，用户必须首先给出矩阵的行数（它同时也是矩阵的列数）。可按图5-11的形式来提示用户输入所需要的数据。

Please enter the number of rows in the maze.

I work on square mazes only.

Enter -1 to terminate the program.

Press <Enter> when done.

图5-11 获取迷宫大小的屏幕显示信息

接下来要求用户逐行输入矩阵数据：

Please Enter Row 1

Press <Enter> when done

每输入完一行数据，都应把它们显示在屏幕上以便于用户验证。为此，有必要提供编辑功能来帮助纠正错误。重新输入整行数据是一种可取的编辑方式（从用户的观点来看）。输入模块需要验证在迷宫的入口或出口处是否有障碍物，如果有障碍物，则不可能存在路径。此外，输入模块应对用户所有可能的输入错误进行校验。在后面的讨论中假定输入模块已经做过这样的验证，并且在入口和出口处没有障碍物。

在设计输入模块的用户接口时还涉及其他一些问题，如颜色和声音的使用；询问信息的显示尺寸；询问信息在屏幕上的显示位置；每输入完一行数据时屏幕是否需要滚动；或者是否需要清除上一行数据并在同一位置显示下一行数据等。

在这里再一次看到，本来看上去很简单的任务（读入一个矩阵）实际上很复杂，因为我们想使它的界面非常友好。

在设计输出模块时所涉及的问题基本上与设计输入模块所考虑的问题相同。

### 3. 开发计划

在前面设计阶段已经指出需要设计四个程序模块。实际上还需要一个根模块来调用这四个模块，调用的次序是：欢迎模块、输入模块、寻找路径模块和输出模块。

程序的模块结构如图5-12所示。每个模块都可以独立地编写。根模块将被编写成一个 main 函数，欢迎模块、输入模块、寻找路径模块和输出路径模块将分别对应于一个函数。

至此，程序如图5-13所示。

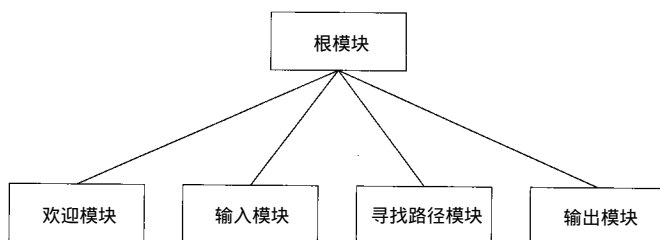


图5-12 迷宫程序的模块结构

```
// 函数模块
// 欢迎模块 Welcome
// 输入模块 InputMaze
// 寻找路径模块 FindPath
// 输出模块 OutputPath
void main(void)
{
    Welcome();
    InputMaze();
    if (FindPath()) OutputPath();
    else cout << "No path" << endl;
}
```

图5-13 迷宫程序的形式

### 4. 程序开发

数据结构和算法的问题仅出现在寻找路径模块的开发过程中。因此，这里仅设计寻找路径模块。练习16要求读者设计出其他的模块。在详细设计寻找路径模块的编码之前，可以先给出图5-14所示的C++伪代码。可以很容易判断这段代码的正确性。遗憾的是，不能直接在计算机中使用这种形式的代码，必须把这种伪代码细化成真正的C++代码。

在试图对图5-14的伪代码进行细化之前，首先探讨如何寻找迷宫路径。首先把迷宫的入口作为当前位置。如果当前位置是迷宫出口，那么已经找到了一条路径，搜索工作结束。如果当

前位置不是迷宫出口，则在当前位置上放置障碍物，以便阻止搜索过程又绕回到这个位置。然后检查相邻的位置中是否有空闲的（即没有障碍物），如果有，就移动到这个新的相邻位置上，然后从这个位置开始搜索通往出口的路径。如果不成功，选择另一个相邻的空闲位置，并从它开始搜索通往出口的路径。为了方便移动，在进入新的相邻位置之前，把当前位置保存在一个堆栈中。如果所有相邻的空闲位置都被探索过，并且未能找到路径，则表明在迷宫中不存在从入口到出口的路径。

```
bool FindPath()
{
    在迷宫中寻找一条通往出口的路径；
    if (找到一条路径) return true;
    else return false;
}
```

图5-14 FindPath的第一个版本

使用上述策略来考察图 5-8 的迷宫。首先把位置 (1,1) 放入堆栈，并从它开始进行搜索。由于位置 (1,1) 只有一个空闲的邻居 (2,1)，所以接下来将移动到位置 (2,1)，并在位置 (1,1) 上放置障碍物，以阻止稍后的搜索再次经过这个位置。从位置 (2,1) 可以移动到 (3,1) 或 (2,2)。假定移动到位置 (3,1)。在移动之前，先在位置 (2,1) 上放置障碍物并将其放入堆栈。从位置 (3,1) 可以移动到 (4,1) 或 (3,2)。假定移动到位置 (4,1)，则在位置 (3,1) 上放置障碍物并将其放入堆栈。从位置 (4,1) 开始可以依次移动到 (5,1)、(6,1)、(7,1) 和 (8,1)。到了位置 (8,1) 以后将无路可走。此时堆栈中包含的路径从 (1,1) 至 (8,1)。为了探索其他的路径，从堆栈中删除位置 (8,1)，然后回退至位置 (7,1)，由于位置 (7,1) 也没有新的、空闲的相邻位置，因此从堆栈中删除位置 (7,1) 并回退至位置 (6,1)。按照这种方式，一直要回退到位置 (3,1)，然后才可以继续移动（即移动到位置 (3,2)）。注意在堆栈中始终包含从入口到当前位置的路径。如果最终到达了出口，那么堆栈中的路径就是所需要的路径。

为了细化图 5-14，需要对迷宫（一个 0 和 1 的矩阵）、迷宫的每个位置以及堆栈进行描述。首先考虑迷宫的描述。迷宫一般被描述成一个 int 类型的二维数组 maze。（由于每个迷宫位置仅有 0 或 1 两种取值，因此可以把 16 个迷宫位置压缩成一个 int 类型的变量，假定每个变量的长度是两个字节。这种压缩可以使迷宫所需的空间减小 16 倍。不过，由于访问每个迷宫位置的难度增加，因此程序的运行时间将随之增加）。迷宫矩阵中的位置 (i, j) 对应于数组 maze 的位置 [i, j]。

对于迷宫内部的位置（非边界位置），有四种可能的移动方向：右、下、左和上。对于迷宫的边界位置，只有两种或三种可能的移动。为了避免在处理内部位置和边界位置时存在差别，可以在迷宫的周围增加一圈障碍物。对于一个  $m \times n$  的迷宫，这一圈障碍物将占据数组 maze 的第 0 行，第  $m+1$  行，第 0 列和第  $m+1$  列（如图 5-15 所示）。

现在迷宫中的所有的位置都处在所添加的障碍物的边界以内，因此，对于原迷宫中的每个位置，都可以有四种可能的移动方向。通过在迷宫周围添加边界，可以使程序不必去处理边界条件，这可以大大简化代码的设计。这种简化是以稍稍增加数组 maze 的空间

```
1 1 1 1 1 1 1 1 1 1 1 1
1 0 1 1 1 1 1 0 0 0 0 1
1 0 0 0 0 0 1 0 1 0 0 1
1 0 0 0 1 0 1 0 0 0 0 1
1 0 1 0 1 0 1 0 1 1 0 1
1 0 1 0 1 0 1 0 1 0 0 1
1 0 1 1 1 0 1 0 1 0 1 1
1 0 1 0 0 0 1 0 1 0 1 1
1 0 1 0 1 1 1 0 1 0 0 1
1 1 0 0 0 0 0 0 1 0 0 1
1 0 0 0 0 1 1 1 1 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1
```

图5-15 为图5-8的迷宫增加一圈障碍物

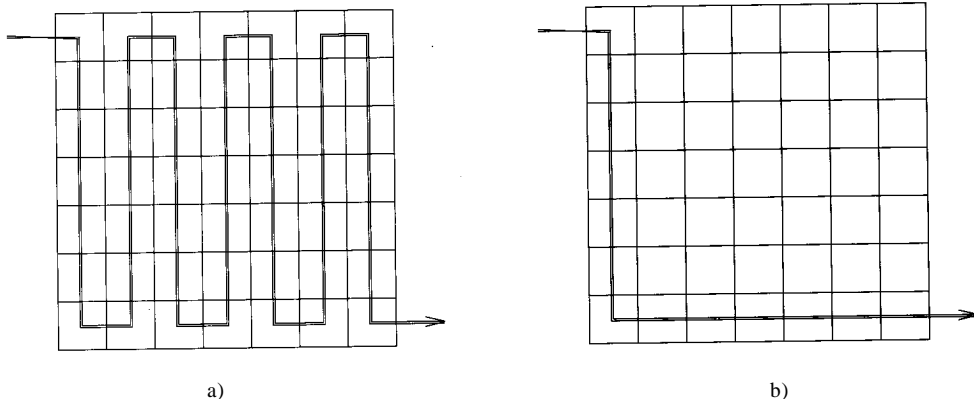


图5-16 没有障碍物的迷宫中的路径

a) 一条长路径 b) 一条短路径

需求为代价的。

可以用行号和列号来指定每个迷宫位置，行号和列号被分别称之为迷宫位置的行坐标和列坐标。可以定义一个相应的类 `Position` 来表示迷宫位置，它有两个私有成员 `row` 和 `col`。可以使用类型为 `Position` 的对象来跟踪迷宫的位置。为了保存从入口到当前位置的路径，可以采用以下基于公式化描述的堆栈：

```
Stack<Position> path(MaxPathLength);
```

其中 `MaxPathLength` 是指最大可能的路径长度（从入口到迷宫中任一位置）。由于在一条路径中所出现的位置各不相同，因此对于一个没有障碍物的  $m \times m$  迷宫，最长的路径可包含  $m^2$  个位置（如图5-16a所示）。由于路径中的最后一个位置不必存储到堆栈中，因此有 `MaxPathLength =  $m^2 - 1$` 。注意，在一个没有障碍物的迷宫中，任意两点之间总是存在一条少于  $2m$  个位置的路径。不过，这里我们并没有保证搜索程序能找到最短路径。

现在可以来细化图5-14，细化的结果见图5-17，它已经接近了 C++ 程序。接下来需要着手解决这样的问题：对于位置 `here`，下一步将移动到它的哪一个相邻位置。如果按一种固定的方式来

```
bool FindPath()
{ // 寻找从位置(1,1)到出口(m,m)的路径
  增加一圈障碍物;

  // 对跟踪当前位置的变量进行初始化
  Position here;
  here.row = 1;
  here.col = 1;

  maze[1][1] = 1; // 阻止返回入口

  // 寻找通往出口的路径
  while (不是出口) do {
    选择一个相邻位置;
    if (存在这样一个相邻位置) {
      把当前位置 here 放入堆栈 path;
      // 移动到相邻位置，并在当前位置放上障碍物
      here = neighbor;
      maze[here.row][here.col] = 1;
    }
    else {
      // 不能继续移动，需回溯
      if (堆栈 path 为空) return false;
      回溯到 path 栈顶中的位置 here;
    }
  }
  return true;
}
```

图5-17 图5-14的细化版本



选择可行的位置，将可以使问题得到简化。例如，可以首先尝试向右移动，然后是向下，向左，最后是向上。一旦做出了选择，就需要知道下一个位置的坐标。可以通过保留一个如图 5-18 所示的偏移量表来计算这些坐标。可以分别把向右、向下、向左和向上移动编号为 0, 1, 2 和 3。在图 5-18 中，offset[i].row 和 offset[i].col 分别给出了从当前位置沿方向 i 移动到下一个相邻位置时，row 和 col 坐标的增量。例如，如果当前处于位置 (3, 4)，则其右边相邻位置的行坐标为 3+offset[0].row=3，列坐标为 4+offset[0].col=5。

把上述细化工作并入图 5-17 的代码之中，就得到了程序 5-13 中的 C++ 代码。5-13 中的代码假定 maze、m(迷宫的大小)和 path 都是按如下方式定义的全局变量：

```
int **maze, m;
```

```
Stack<Position> *path;
```

并且 FindPath 是 Position 的一个友元。变量 maze 和 m 由函数 InputMaze 负责初始化。

移动	方向	offset[move].row	offset[mov].col
0	向右	0	1
1	向下	1	0
2	向左	0	-1
3	向上	-1	0

图 5-18 偏移量表

程序 5-13 搜索迷宫路径的代码

```
bool FindPath ( )
{// 寻找从位置(1,1)到出口(m,m)的路径
//如果成功则返回 true ，否则返回 false
// 如果内存不足则引发异常 NoMem
path = new Stack<Position>(m * m - 1);
//对偏移量进行初始化
Position offset[4];
offset[0].row = 0; offset[0].col = 1; //向右
offset[1].row = 1; offset[1].col = 0; // 向下
offset[2].row = 0; offset[2].col = -1; //向左
offset[3].row = -1; offset[3].col = 0; //向上
//在迷宫周围增加一圈障碍物
for (int i = 0; i <= m+1; i++) {
    maze [0] [i] = maze[m+1] [i] = 1; //底和顶
    maze [i] [0] = maze[i] [m+1] = 1; // 左和右
}
Position here;
here.row = 1;
here.col = 1;
maze [1] [1] = 1; // 阻止返回入口
int option = 0;
int LastOption = 3;
//寻找一条路径
while (here.row != m || here.col != m) {// 不是出口
    //寻找并移动到一个相邻位置
```

```

int r, c;
while (option <= LastOption) {
    r = here.row + offset[option].row;
    c = here.col + offset[option].col;
    if (maze[r][c] == 0) break;
    option++; //下一个选择
}
// 找到一个相邻位置了吗?
if (option <= LastOption) { // 移动到maze[r] [c]
    path->Add(here);
    here.row = r; here.col = c;
    //设置障碍物以阻止再次访问
    maze[r][c] = 1;
    option = 0;
}
else { //没有可用的相邻位置, 回溯
    if (path->IsEmpty()) return false;
    Position next;
    path->Delete(next);
    if (next.row == here.row)
        option = 2 + next.col - here.col;
    else option = 3 + next.row - here.row;
    here = next;
}
}
return true; //到达迷宫出口
}

```

FindPath首先创建一个足够大的堆栈 \*path, 然后对偏移量数组进行初始化, 并在迷宫周围增加一圈障碍物。在 while 循环中, 从当前位置 here 出发, 按下列次序来选择下一个移动位置: 向右、向下、向左和向上。如果能够移动到下一个位置, 则将当前位置放入堆栈 path, 并移动到下一个位置。如果找不到下一个可以移动的位置, 则退回到前一个位置。如果无法回退一个位置 (即堆栈为空), 则表明不存在通往出口的路径。当回退至堆栈的顶部位置 (next) 时, 可以重新选择另一个可能的相邻位置, 这可以利用 next 和 here 来推算。注意 here 是 next 的一个邻居。对下一个移动位置的选择可用以下代码来实现:

```

if ( next.row == here.row)
    Option= 2 + next.col - here.col;
else option = 3 + next.row - here.row;

```

下面分析程序的时间复杂性。可以注意到, 在最坏情况下, 可能要遍历每一个空闲的位置, 而每个位置进入堆栈的机会最多有四次。(每次从一个位置向下一个位置移动时, 该位置都要进入堆栈, 而对于每个位置, 可以有四种可能的移动选择)。因而每个位置从堆栈中被删除的机会也最多有四次。对于每个位置, 需花费  $\Theta(1)$  的时间来检查它的相邻位置。因此, 程序的时间复杂性应为  $O(\text{unblocked})$ , 其中 unblocked 是迷宫中的空闲位置数目。  $O(\text{unblocked}) = O(m^2)$ 。

## 练习

7. 利用归纳法 (对碟子的数目进行归纳) 证明程序 5-6 的正确性。

8. 假定汉诺塔中的碟子按  $1 \sim n$  编号, 最小的碟子为 1 号碟子。修改程序 5-6, 让它同时输出被移动碟子的编号。这种修改只需简单地改动语句 `cout` 即可, 要求不得对其他地方进行修改。

9. 编写程序 5-7 中的 `ShowState` 函数, 假定输出设备为计算机屏幕。注意考虑时间延迟, 以便显示信息不至于变化太快。用不同的颜色显示每个碟子。

10. 当输出设备为计算机屏幕时, 能否在每次移动之后只显示三座塔的状态而不必在内存中明确地存储塔的状态? 试解释。

11. 假定在重排火车车厢问题中用  $k$  个基于公式化描述的堆栈来表示  $k$  个缓冲铁轨。请问每个堆栈应有多大?

\*12. 1) 采用  $k$  个缓冲铁轨来进行车厢重排, 程序 5-8 总能成功吗?

2) 车厢被移动的总次数为:  $n + (\text{车厢被移动到缓冲铁轨的次数})$ 。假定对于给定的初始次序, 使用程序 5-8 和  $k$  个缓冲铁轨能够完成车厢重排。请给出程序 5-8 最少的移动次数, 并证明该结论。

13. 假定每个缓冲铁轨  $i$  中最多只能容纳  $s_i$  节车厢, 其中  $1 \leq i \leq k$ , 请重新编写一个车厢重排程序。

14. 在开关盒布线问题中, 注意到当一个网组的两个针脚都进入堆栈的时候, 处理过程将结束。假定开关盒布线问题的输入是一组网组, 每个网组对应于两个针脚。编写一个 C++ 程序来输入每个网组, 并判断开关盒是否可以布线。要求使用一个新的函数 `CheckBox` 来进行这种判断。如果向堆栈中添加一个针脚时, 堆栈中出现了两个属于同一网组的针脚, 则函数 `CheckBox` 可以立即终止。程序的时间复杂性应为  $\Theta(n)$ , 其中  $n$  为针脚的数目。需要多大的堆栈?

15. 用类 `Chain` 和 `LinkedStack` 解决离线等价类问题, 在生成等价类时, 使用了函数 `Chain::Delete` 来删除节点, 用函数 `LinkedStack::Add` 来创建节点。为了避免这两种操作, 可以编写自己的代码来对链表和堆栈进行插入和删除操作, 要求仅当节点 `chain[j]` 的 `data` 域  $x$  满足 `out[x]=0` 时, 才将该节点加入堆栈。也即, 堆栈中的每个节点任何时候都只会出现在一个链表中。在执行完关系的输入以及数组 `out` 的创建以后, 不必再调用函数 `new`。另外由于在输出所有的等价类以后程序终止运行, 因此可以不必调用函数 `delete`。

1) 编写一个解决离线等价类问题的程序, 要求不使用 `Chain` 类和 `LinkedStack` 类。程序应使用普通的代码 (即自己编写的代码) 来对链表和堆栈进行添加和删除操作, 而且始终不要使用 `delete` 函数, 并且在数组 `out` 创建完以后也不去调用 `new` 函数。程序的复杂性应与元素及关系的数目成正比, 试证明之。

2) 使用适当的测试数据比较本代码及程序 5-12 代码的运行时间。

16. 完成迷宫程序的设计, 编写出以下 C++ 程序:

1) 一个漂亮的 `Welcome` 函数

2) 一个稳定性好的 `InputMaze` 函数。例如, 如果没有足够的内存来创建数组 `maze`, 应能检测出来并输出一个错误信息。另外, 在输入过程中应给出足够的用户提示。

3) 一个 `OutputPath` 函数, 用于输出从入口到出口的路径 (不是从出口到入口)。

使用一些迷宫例子来测试代码。

17. 重新设计迷宫程序, 要求按图 5-8 的形式来显示迷宫, 并且用不同的颜色标出原始障碍物位置、由算法添加的障碍物位置、空闲位置以及从入口到当前位置的路径。 `FindPath` 函数应在每一次移动时都相应地修改显示状态。为了能让用户看清显示过程, 必须使代码减速, 做到大约每秒钟移动一次。为此, 可在代码中插入等待语句, 每次等待一定的时间 (比如 1 秒)。使用一些迷宫例子来测试代码。

18. 修改函数FindPath, 使得从当前位置开始, 可尝试向以下方向的相邻位置移动: 北、东北、东、东南、西和西北。使用适当的迷宫来测试修改后代码的正确性。

19. 对于堆栈path的最大尺寸, 给出一个比 $m^2-1$ 更理想的上限。

20. 由于堆栈path是动态定义的, 因此其大小可以取值为迷宫中空闲位置的数目减 1。按照这种思想对程序 5-13 进行修改。

21. 修改程序 5-13, 用链表形式的堆栈来实现 path。对于迷宫应用来说, 采用链表形式的堆栈与采用公式化描述的堆栈各有哪些优缺点?

22. 在程序 5-13 的代码中, 堆栈应足够大以便满足所有的添加操作。因此, 在类 Stack 的添加操作中没有必要测试堆栈是否已满。同时, 利用 Stack 的成员函数按照从入口到出口的次序输出路径也存在不少困难。用一个普通的堆栈来重写程序 5-13 的代码, 也即使用一个动态定义的一维数组和一个变量 top 来实现堆栈。最终的路径可按照 path 中位置 1 到位置 top 的次序输出。使用迷宫的例子来测试代码。

23. 在迷宫中寻找路径的策略实际上是一个递归的策略。从当前位置寻找并移动到它的一个相邻位置, 然后确定从这个相邻位置到出口之间是否存在一条路径, 如果存在一条路径, 则路径搜索过程结束, 否则必须继续寻找并移动到另一个相邻位置。采用递归方法重写函数 FindPath (见程序 5-13)。采用适当的迷宫例子来测试代码的正确性。

## 5.6 参考及推荐读物

开关盒布线算法选自如下论文:

1) C.Hsu. General River Routing Algorithm. *ACM/IEEE Design Automation Conference*, 578~583, 1983。

2) R.Pinter. River-routing: Methodology and Analysis. *Third Caltech Conference on VLSI*, 1983.3。

## 第6章 队 列

像堆栈一样，队列也是一种特殊的线性表。队列的插入和删除操作分别在线性表的两端进行，因此，队列是一个先进先出（first-in-first-out, FIFO）的线性表。尽管可以很容易地从线性表类LinearList（见程序3-1）和链表类Chain（见程序3-8）中派生出队列类，但在本章中并没有这样做。出于对执行效率的考虑，我们把队列设计成一个基类，分别采用了公式化描述和链表描述。

在本章的应用部分，给出了四个使用队列的应用。第一个应用是关于5.5.3节所介绍的火车车厢重排问题。在本章中对这个问题做了修改，要求缓冲铁轨按FIFO方式而不是LIFO方式工作；第二个应用是关于寻找两个给定点之间最短路径的问题，这是一个经典的问题。可以把这个应用看成是5.5.6节迷宫问题的一种变化，即寻找从迷宫入口到迷宫出口的最短路径。5.5.6节中的代码并不能保证得到一条最短的路径，它只能保证如果存在一条从入口到出口的路径，则一定能找到这样一条路径（没有限定长度）；第三个应用选自计算机视觉领域，主要用于识别图像中的图元；最后一个应用是一个工厂仿真程序。工厂内有若干台机器，每台机器能够执行一道不同的工序。每一项任务都由一系列工序组成。我们给出了一个仿真程序，它能够仿真工厂中的任务流。该程序能够确定每项任务所花费的总的等待时间以及每台机器所产生的总的等待时间，可以根据这些信息来改进工厂的设计。

为了获得较高的执行效率，本章中每个应用都采用了队列数据结构。在后续章节中还会介绍其他几种队列应用。

### 6.1 抽象数据类型

**定义** [队列] 队列（queue）是一个线性表，其插入和删除操作分别在表的不同端进行。添加新元素的那一端被称为队尾（rear），而删除元素的那一端被称为队首（front）。

一个三元素的队列如图6-1a所示，从中删除第一个元素A之后将得到图6-1b所示的队列。如果要向图6-1b的队列中添加一个元素D，必须把它放在元素C的后面。添加D以后所得到的结果如图6-1c所示。

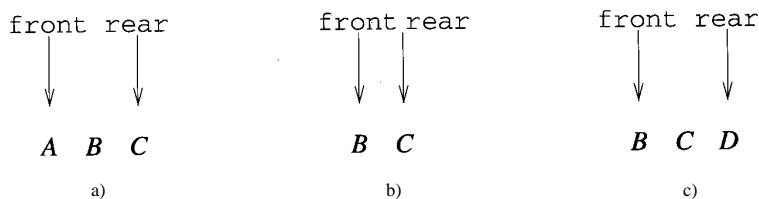


图6-1 队列举例

所以，队列是一个先进先出（FIFO）的线性表，而堆栈是一个先进后出（LIFO）的线性表。队列的抽象数据类型描述见ADT 6-1。

## ADT6-1 队列的抽象数据类型描述

抽象数据类型 *Queue* {

实例

有序线性表，一端称为 front，另一端称为 rear；

操作

*Create*(): 创建一个空的队列；*IsEmpty*(): 如果队列为空，则返回 true，否则返回 false；*IsFull*(): 如果队列满，则返回 true；否则返回 false；*First*(): 返回队列的第一个元素；*Last*(): 返回队列的最后一个元素；*Add* (x): 向队列中添加元素 *x*；*Delete* (x): 删除队首元素，并送入 *x*；

}

## 6.2 公式化描述

假定采用公式 (6-1) 来描述一个队列。

$$location(i) = i - 1 \quad (6-1)$$

这个公式在公式化描述的堆栈中工作得很好。如果使用公式 (6-1) 把数组 `queue[MaxSize]` 描述成一个队列，那么第一个元素为 `queue[0]`，第二个元素为 `queue[1]`，…。`front` 总是为 0，`rear` 始终是最后一个元素的位置，队列的长度为 `rear+1`。对于一个空队列，有 `rear = -1`。使用公式 (6-1)，图 6-1 中的队列可以表示成图 6-2 的形式。

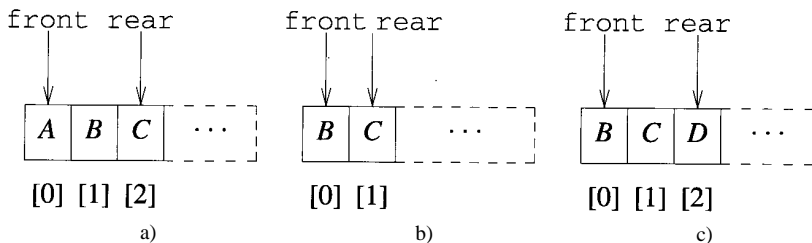


图 6-2 使用公式 (6-1) 描述图 6-1 中的队列

向队列中添加一个元素时，需要把 `rear` 增 1，并把新元素放入 `queue[rear]`。这意味着一次添加操作所需要的时间为  $O(1)$ 。删除一个元素时，把位置 1 至位置 `n` 的元素分别左移一个位置，因此删除一个元素所花费的时间为  $\Theta(n)$ ，其中 `n` 为删除完成之后队列中的元素数。如此看来，公式 (6-1) 应用于堆栈，可使堆栈的插入和删除操作均耗时  $\Theta(1)$ ，而应用于队列，则使队列的删除操作所需要的时间达到  $\Theta(n)$ 。

如果采用公式 (6-2)，就可以使队列的删除操作所需要的时间减小至  $\Theta(1)$ 。

$$location(i) = location(1) + i - 1 \quad (6-2)$$

从队列中删除一个元素时，公式 (6-2) 不要求把所有的元素都左移一个位置，只需简单地把 `location(1)` 增加 1 即可。图 6-3 给出了在使用公式 (6-2) 时，图 6-1 中各队列的相应描述。注意，在使用公式 (6-2) 时，`front = location(1)`，`rear = location(最后一个元素)`，一个空队列

具有性质  $\text{rear} < \text{front}$ 。

如图6-3b所示,每次删除操作将导致  $\text{front}$  右移一个位置。当  $\text{rear} < \text{MaxSize}-1$  时才可以直接在队列的尾部添加新元素。若  $\text{rear} = \text{MaxSize}-1$  且  $\text{front} > 0$  时(表明队列未滿),为了能够继续向队列尾部添加元素,必须将所有元素平移到队列的左端(如图6-4所示),以便在队列的右端留出空间。对于使用公式(6-1)的队列来说,这种平移操作将使最坏情况下的时间复杂性增加  $O(1)$ ,而对于使用公式(6-2)的队列来说,最坏情况下的时间复杂性则增加了  $O(n)$ 。所以,使用公式(6-2)在提高删除操作执行效率的同时,却降低了添加操作的执行效率。

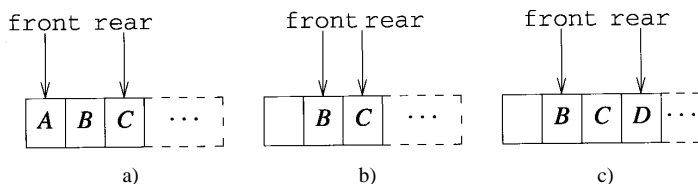


图6-3 使用公式(6-2)描述图6-1中的队列

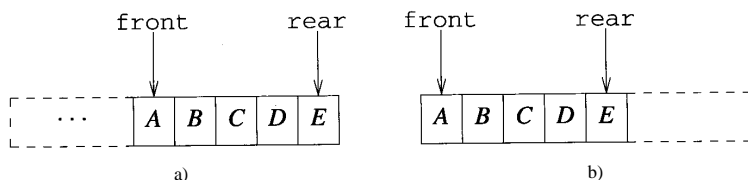


图6-4 队列的平移

a) 移位之前 b) 移位之后

若使用公式(6-3),则队列的添加和删除操作在最坏情况下的时间复杂性均变成  $O(1)$ 。

$$\text{location}(i) = (\text{location}(1) + i - 1) \% \text{MaxSize} \quad (6-3)$$

这时,用来描述队列的数组被视为一个环(如图6-5所示)。在这种情况下,对  $\text{front}$  的约定发生了变化,它指向队列首元素的下一个位置(逆时针方向),而  $\text{rear}$  的含义不变。向图6-5a中的队列添加一个元素将得到图6-5b所示的队列,而从图6-5b的队列中删除一个元素则得到图6-5c所示的队列。

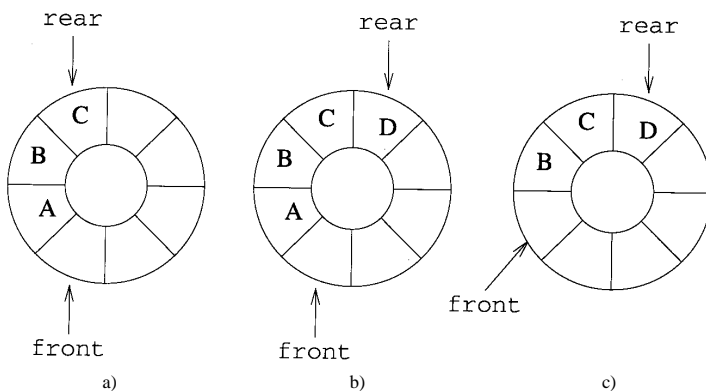


图6-5 循环队列

a) 初始状态 b) 添加 c) 删除



当且仅当  $\text{front}=\text{rear}$  时队列为空。初始条件  $\text{front}=\text{rear}=0$  定义了一个初始为空的队列。现在需要确定队列为满的条件。如果不断地向图 6-5b 的队列添加元素,直到队列满为止,那么将看到图 6-6所示的情形。这时有  $\text{front}=\text{rear}$ ,竟然与队列为空的条件完全一样!因此,我们无法区分出队列是空还是满。为了避免这个问题,可以不允许队列被填满。为此,在向队列添加一个元素之前,先判断一下本次操作是否会导致队列被填满,如果是,则报错。因此,队列的最大容量实际上是  $\text{MaxSize}-1$ 。

可用程序 6-1 所示的 C++ 类来实现抽象数据类型 Queue。在实现公式化描述的堆栈时(见程序 5-1),为了简化代码的设计,重用了 LinearList 类(见程序 3-1)的定义。然而不能通过使用同样的方法来实现 Queue 类,因为 Queue 的实现基于公式(6-3),而 LinearList 的实现基于公式(6-1)。程序 6-2 和程序 6-3 给出了 Queue 成员函数的代码。注意观察 Queue 的构造函数是怎样保证循环队列的容量比数组的容量少 1 的。利用如下语句,可以创建一个能够容纳 12 个整数的队列:

```
Queue<int> Q(12);
```

队列的成员函数与堆栈的对应函数相类似,因此这里不再具体介绍这些函数。当 T 是一个内部数据类型时,队列构造函数和析构函数的复杂性均为  $\Theta(1)$ ;而当 T 是一个用户定义的类时,构造函数和析构函数的复杂性均为  $O(\text{MaxStackSize})$ 。其他队列操作的复杂性均为  $\Theta(1)$ 。

程序 6-1 公式化类 Queue

```
template<class T>
class Queue {
// FIFO 对象
public:
    Queue(int MaxQueueSize = 10);
    ~Queue() {delete [] queue;}
    bool IsEmpty() const {return front == rear;}
    bool IsFull() const {return (
        ((rear + 1) % MaxSize == front) ? 1 : 0);}
    T First() const; //返回队首元素
    T Last() const; // 返回队尾元素
    Queue<T>& Add(const T& x);
    Queue<T>& Delete(T& x);
private:
    int front; //与第一个元素在反时针方向上相差一个位置
    int rear; // 指向最后一个元素
    int MaxSize; // 队列数组的大小
    T *queue; // 数组
};
```

程序 6-2 Queue 类的成员函数

```
template<class T>
```

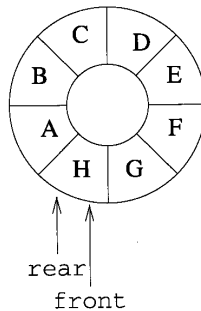


图 6-6 能容纳  $\text{MaxSize}$  个元素的循环队列

```
Queue<T>::Queue(int MaxQueueSize)
{// 创建一个容量为 MaxQueueSize的空队列
    MaxSize = MaxQueueSize + 1;
    queue = new T[MaxSize];
    front = rear = 0;
}

template<class T>
T Queue<T>::First() const
{// 返回队列的第一个元素
// 如果队列为空,则引发异常 OutOfBounds
    if (IsEmpty()) throw OutOfBounds();
    return queue[(front + 1) % MaxSize];
}

template<class T>
T Queue<T>::Last() const
{// 返回队列的最后一个元素
// 如果队列为空,则引发异常 OutOfBounds
    if (IsEmpty()) throw OutOfBounds();
    return queue[rear];
}
```

程序6-3 Queue类的成员函数

```
template<class T>
Queue<T>& Queue<T>::Add(const T& x)
{// 把 x 添加到队列的尾部
// 如果队列满,则引发异常 NoMem
    if (IsFull()) throw NoMem();
    rear = (rear + 1) % MaxSize;
    queue[rear] = x;
    return *this;
}

template<class T>
Queue<T>& Queue<T>::Delete(T& x)
{// 删除第一个元素,并将其送入 x
// 如果队列为空,则引发异常 OutOfBounds
    if (IsEmpty()) throw OutOfBounds();
    front = (front + 1) % MaxSize;
    x = queue[front];
    return *this;
}
```

## 练习

1. 扩充队列的ADT,增加以下函数:
  - 1) 确定队列中的元素数目。

2) 输入一个队列。

3) 输出一个队列。

2. 扩充队列的ADT，增加以下函数：

1) 把一个队列分解成两个队列，其中一个队列包含原队列中的第 1、3、5、... 个元素，另一个队列包含了其余元素。

2) 合并两个队列（称队列1和队列2），在新队列中，从队列1开始，两个队列的元素轮流排列，若某个队列中的元素先用完，则将另一个队列中的剩余元素依次添加在新队列的尾部。合并完成后，各元素之间的相对次序应与合并前的相对次序相同。

请扩充公式化描述的队列类的定义，增加以上两个成员函数。编写并测试代码。

3. 使用公式（6-2）设计一个相应的C++队列类，编写并测试所有代码。

4. 修改程序6-1中的Queue类，使得队列的容量与数组queue的大小相同。为此，可引入另外一个私有成员LastOp来跟踪最后一次队列操作。可以肯定，如果最后一次队列操作为Add，则队列一定不为空；如果最后一次队列操作为Delete，则队列一定不会满。因此，当front=rear时，可使用LastOp来区分一个队列是空还是满。试测试修改后的代码。

5. 双端队列（deque）是指这样一个有序线性表：可在表的任何一端进行插入和删除操作。

1) 给出双端队列的抽象数据类型描述，要求包含以下操作：*Create*，*IsEmpty*，*IsFull*，*Left*，*Right*，*AddLeft*，*AddRight*，*DeleteLeft*和*DeleteRight*。

2) 采用公式（6-3）来描述双端队列。设计一个与双端队列抽象数据类型描述相对应的C++类Deque，要求编写出所有类成员的代码。

3) 采用适当的测试数据来测试所编写的代码。

## 6.3 链表描述

像堆栈一样，也可以使用链表来实现一个队列。此时需要两个变量front和rear来分别跟踪队列的两端，这时有两种可能的情形：从front开始链接到rear（如图6-7a所示）或从rear开始链接到front（如图6-7b所示）。不同的链接方向将使添加和删除操作的难易程度有所不同。图6-8和6-9分别演示了添加元素和删除元素的过程。可以看到，两种链接方向都很适合于添加操作，而从front到rear的链接更便于删除操作的执行。因此，我们将采用从front到rear的链接模式。

可以取初值front=rear=0，并且认定当且仅当队列为空时front=0。利用3.4.3节的扩展，可以把类LinkedList定义为Chain类（见程序3-8）的一个派生类，练习6即按照这种方式来实现LinkedList。在本节中把LinkedList定义为一个基类。

程序6-4给出了一个链表队列的类定义，程序6-5和程序6-6给出了相应的成员函数。Node类与自定义链表堆栈（见程序5-4）中所使用的相同。为了便于成员函数的实现，可把LinkedList定义为Node的友元。可以分别采用空队列、单元素队列和多元素队列来人工跟踪LinkedList代码的执行。除析构函数外，链表队列所有成员函数的复杂性均为 $\Theta(1)$ 。

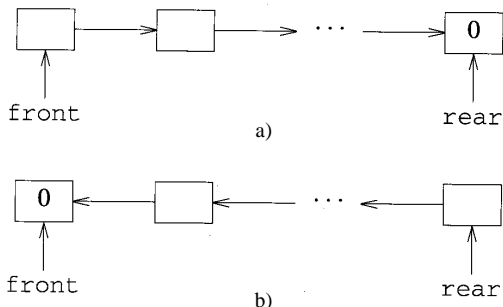


图6-7 链表队列

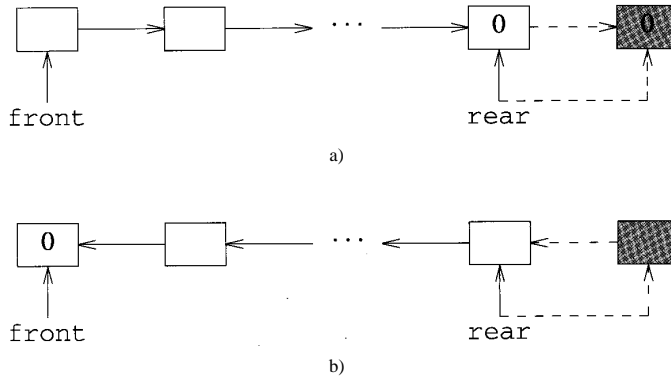


图6-8 向链表队列中添加元素

a) 向图6-7a 的队列添加元素 b) 向图6-7b 的队列添加元素

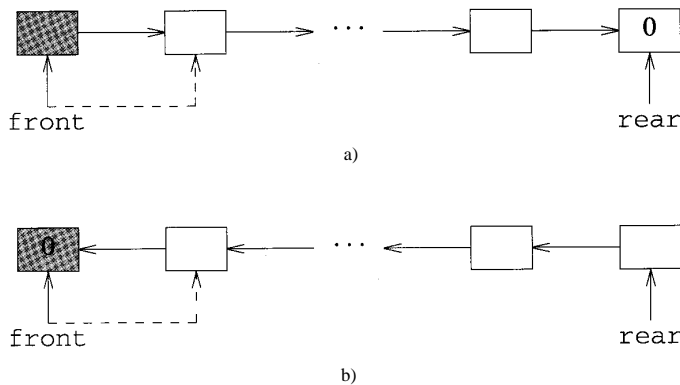


图6-9 从链表队列中删除元素

a) 从图6-7a 的队列中删除元素 b) 从图6-7b 的队列中删除元素

程序6-4 链表队列的类定义

```

template<class T>
class LinkedQueue {
// FIFO对象
public:
    LinkedQueue() {front = rear = 0;} // 构造函数
    ~LinkedQueue(); // 析构函数
    bool IsEmpty() const
        {return ((front) ? false : true);}
    bool IsFull() const;
    T First() const; // 返回第一个元素
    T Last() const; // 返回最后一个元素
    LinkedQueue<T>& Add(const T& x);
    LinkedQueue<T>& Delete(T& x);
private:
    Node<T> *front; // 指向第一个节点
    Node<T> *rear; // 指向最后一个节点
};

```

程序6-5 链表队列的函数实现

```
template<class T>
LinkedList<T>::~~LinkedList()
{
    // 队列析构函数，删除所有节点
    Node<T> *next;
    while (front) {
        next = front->link;
        delete front;
        front = next;
    }
}

template<class T>
bool LinkedList<T>::IsFull() const
{
    // 判断队列是否已满
    Node<T> *p;
    try {p = new Node<T>;
        delete p;
        return false;}
    catch (NoMem) {return true;}
}

template<class T>
T LinkedList<T>::First() const
{
    // 返回队列的第一个元素
    // 如果队列为空，则引发异常 OutOfBounds
    if (IsEmpty()) throw OutOfBounds();
    return front->data;
}

template<class T>
T LinkedList<T>::Last() const
{
    // 返回队列的最后一个元素
    // 如果队列为空，则引发异常 OutOfBounds
    if (IsEmpty()) throw OutOfBounds();
    return rear->data;
}
```

程序6-6 链表队列的函数实现

```
template<class T>
LinkedList<T>& LinkedList<T>::Add(const T& x)
{
    // 把 x 添加到队列的尾部
    // 不捕获可能由 new 引发的 NoMem 异常

    // 为新元素创建链表节点
    Node<T> *p = new Node<T>;
    p->data = x;
    p->link = 0;
```

```

// 在队列尾部添加新节点
if (front) rear->link = p; //队列不为空
else front = p;           // 队列为空
rear = p;

return *this;
}

template<class T>
LinkedQueue<T>& LinkedQueue<T>::Delete(T& x)
{
    // 删除第一个元素，并将其放入 x
    // 如果队列为空，则引发异常 OutOfBounds

    if (IsEmpty()) throw OutOfBounds();

    //保存第一个节点中的元素
    x = front->data;

    // 删除第一个节点
    Node<T> *p = front;
    front = front->link;
    delete p;

    return *this;
}

```

## 练习

6. 利用3.4.3节Chain类的扩充版本（包含函数 Append），从Chain类中派生出链表队列类 LinkedQueue，并设计出相应的成员函数。
7. 采用链表队列来完成练习1。
8. 采用链表队列来完成练习2，所不同的是，要求各操作均就地进行而不得使用新节点。在分解/合并操作完成之后，原输入队列应为空。
9. 采用链表来完成练习5。分别指出每种操作的复杂性。
10. 采用双向链表来完成练习5。分别指出每种操作的复杂性。

## 6.4 应用

### 6.4.1 火车车厢重排

下面来重新考察一下5.5.3节的火车车厢重排问题。如图6-10所示，假定缓冲铁轨位于入轨和出轨之间。由于这些缓冲铁轨均按FIFO的方式运作，因此可将它们视为队列。与5.5.3节一样，禁止将车厢从缓冲铁轨移动至入轨，也禁止从出轨移动车厢至缓冲铁轨。所有的车厢移动都按照图6-10中箭头所示的方向进行。

铁轨 $H_k$ 为可直接将车厢从入轨移动到出轨的通道。因此，可用来容留车厢的缓冲铁轨的数目为 $k-1$ 。

假定重排9节车厢，其初始次序为5, 8, 1, 7, 4, 2, 9, 6, 3，同时令 $k=3$ 。3号车厢不能直接移

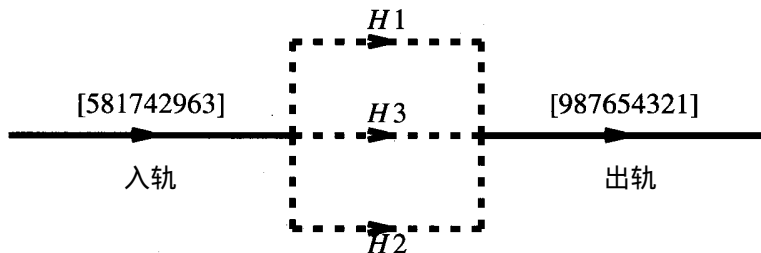


图6-10 三个缓冲铁轨示例

动到出轨，因为1号车厢和2号车厢必须排在3号车厢之前。因此，把3号车厢移动至 $H1$ 。6号车厢可放在 $H1$ 中3号车厢之后，因为6号车厢将在3号车厢之后输出。此后9号车厢可以继续放在 $H1$ 中6号车厢之后，而接下来的2号车厢不可放在9号车厢之后，因为2号车厢必须在9号车厢之前输出。因此，应把2号车厢放在 $H2$ 的首部。之后4号车厢被放在 $H2$ 中2号车厢之后，7号车厢又被放在4号车厢之后。至此，1号车厢可通过 $H3$ 直接移动至出轨，然后从 $H2$ 移动2号车厢至出轨，从 $H1$ 移动3号车厢至出轨，从 $H2$ 移动4号车厢至出轨。由于5号车厢此时仍位于入轨之中，所以把8号车厢移动至 $H2$ ，这样就可以把5号车厢直接从入轨移动至出轨。这之后，可依次从缓冲铁轨中输出6号、7号、8号和9号车厢。

在把一节车厢移动到缓冲铁轨中时，可以采用如下的原则来确定应该把这节车厢移动到哪一个缓冲铁轨。车厢 $c$ 应移动到这样的缓冲铁轨中：该缓冲铁轨中现有各车厢的编号均小于 $c$ ；如果有多个缓冲铁轨都满足这一条件，则选择一个左端车厢编号最大的缓冲铁轨；否则选择一个空的缓冲铁轨（如果有的话）。

### 1. 第一种实现方法

可以采用链表队列来实现车厢重排算法，其中，用链表队列来表示 $k-1$ 个缓冲铁轨。可以按照程序5-8、程序5-9和程序5-10的模式来设计该算法。程序6-7给出了函数Output和Hold的新代码。对于程序5-8中的函数Railroad，应做以下修改：1) 将 $k$ 减1；2)  $H$ 的类型修改为 $\text{LinkedListQueue}<\text{int}> *$ ；3) 把MinS改为MinQ；4) 从Hold的调用中删除最后一个参数( $n$ )。完成车厢重排所需要的时间为 $O(nk)$ 。借助于AVL树（见第11章），可以把复杂性减小至 $O(n\log k)$ 。

程序6-7 使用队列来重排车厢

```
void Output(int& minH, int& minQ, LinkedListQueue<int> H[], int k, int n)
{//从缓冲铁轨移动到出轨，并修改 minH 和 minQ.
    int c; // 车厢编号

    // 从队列 minQ 中删除编号最小的车厢 minH
    H[minQ].Delete(c);
    cout << "Move car " << minH << " from holding track " << minQ << " to output" << endl;

    // 通过检查所有队列的首部，寻找新的 minH和minQ
    minH = n + 2;
    for (int i = 1; i <= k; i++)
        if (!H[i].IsEmpty() &&
            (c = H[i].First()) < minH) {
            minH = c;
```



```

        minQ = i;}
    }

    bool Hold(int c, int& minH, int &minQ, LinkQueue<int> H[], int k)
    { //把车厢c 移动到缓冲铁轨中
      // 如果没有可用的缓冲铁轨，则返回 false，否则返回 true

      // 为车厢 c 寻找最优的缓冲铁轨
      // 初始化
      int BestTrack = 0, // 目前最优的铁轨
          BestLast = 0, // BestTrack 中最后一节车厢
          x;             // 车厢编号

      // 扫描缓冲铁轨
      for (int i = 1; i <= k; i++)
      {
        if (!H[i].IsEmpty()) { // 铁轨 i 不为空
          x = H[i].Last();
          if (c > x && x > BestLast) { // 铁轨 i 尾部的车厢编号较大
            BestLast = x;
            BestTrack = i;
          }
        }
        else // 铁轨i 为空
          if (!BestTrack) BestTrack = i;
      }

      if (!BestTrack) return false; // 没有可用的铁轨

      // 把c 移动到最优铁轨
      H[BestTrack].Add(c);
      cout << "Move car " << c << " from input " << "to holding track " << BestTrack << endl;

      // 如果有必要，则修改 minH和minQ
      if (c < minH) {minH = c; minQ = BestTrack;}

      return true;
    }

```

## 2. 第二种实现方法

如果只是为了简单地输出车厢重排过程中所必要的车厢移动次序，那么只需了解每个缓冲铁轨的最后一个成员是谁以及每节车厢当前位于哪个铁轨即可。如果缓冲铁轨  $i$  为空，则令  $last[i]=0$ ，否则令  $last[i]$  为铁轨中最后一节车厢的编号。如果车厢  $i$  位于入轨之中，令  $track[i]=0$ ；否则，令  $track[i]$  为车厢  $i$  所在的缓冲铁轨。在起始时有  $last[i]=0, 1 \leq i < k, track[i]=0, 1 \leq i \leq n$ 。程序6-8中并未使用队列，它所产生的输出与程序 6-7所产生的输出完全相同，二者均具有相同的渐进复杂性。

程序6-8 不使用队列来重排车厢

```

void Output(int NowOut, int Track, int& Last)
{ //将车厢NowOut 从缓冲铁轨移动到出轨，并修改 Last

```

```
cout << "Move car " << NowOut << " from holding track " << Track << " to output" << endl;
if (NowOut == Last) Last = 0;
}
```

```
bool Hold(int c, int last[], int track[], int k)
```

```
{//把车厢c 移动到缓冲铁轨中
```

```
// 如果没有可用的缓冲铁轨，则返回 false，否则返回true
```

```
// 为车厢 c 寻找最优的缓冲铁轨
```

```
// 初始化
```

```
int BestTrack = 0, // 目前最优的铁轨
```

```
BestLast = 0, // BestTrack中最后一节车厢
```

```
// 扫描缓冲铁轨
```

```
for (int i = 1; i <= k; i++) // find best track
```

```
if (last[i]) { // 铁轨 i 不为空
```

```
if (c > last[i] && last[i] > BestLast) { // 铁轨 i 尾部的车厢编号较大
```

```
BestLast = last[i];
```

```
BestTrack = i;
```

```
}
```

```
else // 铁轨 i 为空
```

```
if (!BestTrack) BestTrack = i;
```

```
if (!BestTrack) return false; // 没有可用的铁轨
```

```
// 把c 移动到最优铁轨
```

```
track[c] = BestTrack;
```

```
last[BestTrack] = c;
```

```
cout << "Move car " << c << " from input " << "to holding track " << BestTrack << endl;
```

```
return true;
```

```
}
```

```
bool Railroad(int p[], int n, int k)
```

```
{// 用k 个缓冲铁轨进行车厢重排，车厢的初始次序为 p[1:n]
```

```
// 如果重排成功，则返回 true，否则返回false
```

```
// 如果空间不足，则引发异常 NoMem
```

```
// 对数组last和track进行初始化
```

```
int *last = new int [k + 1];
```

```
int *track = new int [n + 1];
```

```
for (int i = 1; i <= k; i++)
```

```
last[i] = 0; // 铁轨 i 为空
```

```
for (int i = 1; i <= n; i++)
```

```
track[i] = 0;
```

```
k--; // 铁轨k 作为直接移动的通道
```

```
// 对欲输出的下一节车厢的编号置初值
```

```

int NowOut = 1;

//按序输出车厢
for (int i = 1; i <= n; i++)
    if (p[i] == NowOut) { // 直接输出
        cout << "Move car " << p[i] << " from input to output" << endl;
        NowOut++;

        // 从缓冲铁轨中输出
        while (NowOut <= n && track[NowOut]) { Output(NowOut, track[NowOut], last[NowOut]);
            NowOut++;
        }
    }
    else { // 把车厢 p[i] 移动到缓冲铁轨
        if (!Hold(p[i], last, track, k))
            return false;
    }

return true;
}

```

#### 6.4.2 电路布线

在5.5.6节中，对迷宫老鼠问题的解决方案并不能保证找到一条从迷宫入口到迷宫出口的最短路径。而借助于队列，可以找到这样的路径（如果有的话）。在迷宫中寻找最短路径的问题也存在于其他许多领域。例如，在解决电路布线问题时，一种很常用的方法就是在布线区域叠上一个网格，该网格把布线区域划分成  $n \times m$  个方格，就像迷宫一样，如图 6-11a 所示。从一个方格  $a$  的中心点连接到另一个方格  $b$  的中心点时，转弯处必须采用直角，如图 6-11b 所示。如果已经有某条线路经过一个方格，则封锁该方格。我们希望使用  $a$  和  $b$  之间的最短路径来作为布线的路径，以便减少信号的延迟。

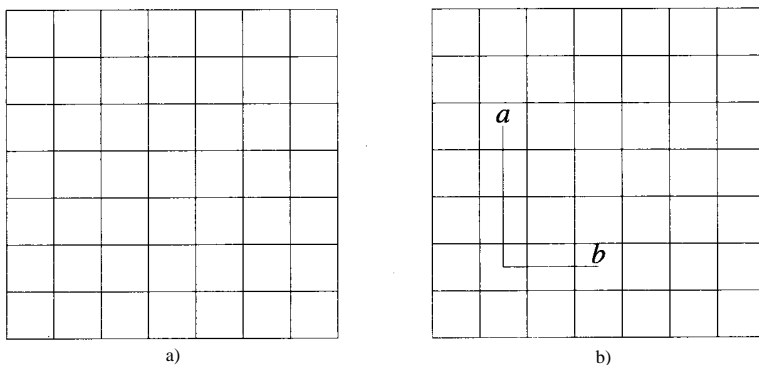


图6-11 电路布线示例

a)  $7 \times 7$  网格 b)  $a$  与  $b$  之间的电线

在下面的讨论中，我们假定你对 5.5.6 节所介绍的迷宫求解算法已经很熟悉。如果不熟悉，

在继续阅读之前，应该认真复习一下5.5.6节的内容。为了找到网格中位置 $a$ 和 $b$ 之间的最短路径，先从位置 $a$ 开始搜索，把 $a$ 可达到的相邻方格都标记为1（表示与 $a$ 相距为1），然后把标号为1的方格可达到的相邻方格都标记为2（表示与 $a$ 相距为2），继续进行下去，直到到达 $b$ 或者找不到可达到的相邻方格为止。图6-12a 演示了这种搜索过程，其中 $a=(3,2)$ ， $b=(4,6)$ 。图中的阴影方格都是被封锁的方格。

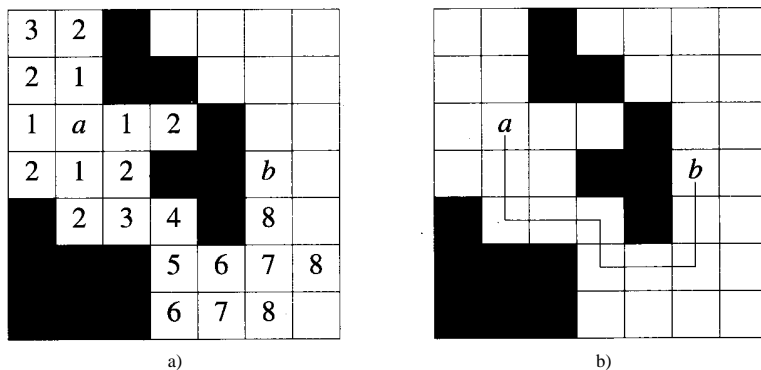


图6-12 电路布线

a) 标识间距 b) 电线路径

按照上述搜索过程，当我们到达 $b$ 时，就可以在 $b$ 上标出 $b$ 与 $a$ 之间的距离，在图6-12a 中， $b$ 上的标号为9。为了得到 $a$ 与 $b$ 之间的最短路径，从 $b$ 开始，首先移动到一个比 $b$ 的编号小的相邻位置上。一定存在这样的相邻位置，因为任一个方格上的标号与它相邻方格上的标号都至少相差1。在图6-12a 中，可从 $b$ 移动到(5,6)。接下来，从当前位置开始，继续移动到比当前标号小1的相邻位置上，重复这个过程，直至到达 $a$ 为止。在图6-12a 的例子中，从(5,6)移动到(6,6)，(6,4)，(5,4)，…。图6-12b 给出了所得到的路径。

现在来看看怎样实现上述策略，以设计出搜索最短路径的 C++代码。我们将从5.5.6节的迷宫解决方案中吸取很多思想。一个 $m \times m$ 的网格被描述成一个二维数组，其中用0表示空白的位置，1表示被封锁的位置。整个网格被包围在一堵由1构成的“墙”中。数组offsets用来帮助我们从一个位置移动到其相邻位置。用一个链表队列来跟踪这样的方格：该方格本身已被编号，而它的相邻位置尚未被编号。也可以采用公式化队列，所不同的是必须估计队列的最大长度，就像在求解迷宫问题时估计堆栈的大小一样。见程序6-9。

程序6-9 寻找电路布线最短路径

```
bool FindPath(Position start, Position finish, int& PathLen, Position * &path)
{//寻找从 start到finish的路径
// 如果成功，则返回 true，否则返回 false
// 如果空间不足，则引发异常 NoMem

if ((start.row == finish.row) &&
    (start.col == finish.col))
    {PathLen = 0; return true;} // start = finish

// 初始化包围网格的“围墙”
```

```

for (int i = 0; i <= m+1; i++) {
    grid[0][i] = grid[m+1][i] = 1; // 底和顶
    grid[i][0] = grid[i][m+1] = 1; // 左和右
}

// 初始化offset
Position offset[4];
offset[0].row = 0; offset[0].col = 1; // 右
offset[1].row = 1; offset[1].col = 0; // 下
offset[2].row = 0; offset[2].col = -1; // 左
offset[3].row = -1; offset[3].col = 0; // 上

int NumOfNbrs = 4; // 一个网格位置的相邻位置数
Position here, nbr;
here.row = start.row;
here.col = start.col;
grid[start.row][start.col] = 2; // 封锁

// 标记可到达的网格位置
LinkedList<Position> Q;
do { // 标记相邻位置
    for (int i = 0; i < NumOfNbrs; i++) {
        nbr.row = here.row + offset[i].row;
        nbr.col = here.col + offset[i].col;
        if (grid[nbr.row][nbr.col] == 0) { // unlabeled nbr, label it
            grid[nbr.row][nbr.col] = grid[here.row][here.col] + 1;
            if ((nbr.row == finish.row) && (nbr.col == finish.col)) break; // 完成
            Q.Add(nbr); // if 结束
        } // for 结束

        //已到达finish吗?
        if ((nbr.row == finish.row) &&
            (nbr.col == finish.col)) break; // 完成

        // 未到达finish, 可移动到nbr吗?
        if (Q.IsEmpty()) return false; // 没有路径
        Q.Delete(here); // 到下一位置
    } while(true);

    // 构造路径
    PathLen = grid[finish.row][finish.col] - 2;
    path = new Position [PathLen];

    // 回溯至 finish
    here = finish;
    for (int j = PathLen-1; j >= 0; j--) {
        path[j] = here;
        // 寻找前一个位置
    }
}

```

```

for (int i = 0; i < NumOfNbrs; i++) {
    nbr.row = here.row + offset[i].row;
    nbr.col = here.col + offset[i].col;
    if (grid[nbr.row][nbr.col] == j+2) break;
}
here = nbr; // 移动到前一个位置
}

return true;
}

```

在程序 6-9 的代码中，假定起始位置和结束位置均未被封锁。程序首先检查 *start* 和 *finish* 是否相同，如果相同，则路径长度为 0，程序终止。否则设置一堵由封锁位置构成的“围墙”，把网格包围起来。然后对 *offset* 数组进行初始化，并在起始位置上标记 2。（所有标号都增加了 2，因为数组中采用 0 和 1 来表示空白位置和封锁位置，为了得到如图 6-12a 所示的标号，必须把程序中的每个标号减去 2）。借助于队列 *Q* 并从位置 *start* 开始，首先移动到与 *start* 相距为 1 的网格位置，然后移动到与 *start* 相距为 2 的网格位置，不断进行下去，直到到达位置 *finish* 或者无法继续移动到一个新的、空白的位置。在后一种情况下，将不存在到达位置 *finish* 的路径，而在前一种情况下，位置 *finish* 将得到一个相应的编号，

如果到达了位置 *finish*，则可以利用网格上的标号来重构路径。路径上的位置（*start* 除外）均被存储在数组 *path* 之中。

由于任意一个网格位置都至多在队列中出现 1 次，所以完成网格编号过程需耗时  $O(m^2)$ （对一个  $m \times m$  的网格来说）。而重构路径的过程需耗时  $O(PathLen)$ ，其中 *PathLen* 为最短路径的长度。

### 6.4.3 识别图元

数字化图像是一个  $m \times m$  的像素矩阵。在单色图像中，每个像素的值要么为 0，要么为 1，值为 0 的像素表示图像的背景，而值为 1 的像素则表示图元上的一个点，我们称其为图元像素。如果一个像素在另一个像素的左侧、上部、右侧或下部，则称这两个像素为相邻像素。识别图元就是对图元像素进行标记，当且仅当两个像素属于同一图元时，它们的标号相同。

考察图 6-13a，其中给出了一个  $7 \times 7$  图像。空白方格代表背景像素，而标记为 1 的方格则代表图元像素。像素 (1,3) 和 (2,3) 属于同一图元，因为它们是相邻的。像素 (2,4) 与 (2,3) 是相邻的，它们也同样属于同一图元，因此，三个像素 (1,3)，(2,3) 和 (2,4) 属于同一图元。由于没有其他的像素与这三个像素相邻，因此这三个像素定义了一个图元。图 6-13a 的图像中存在 4 个图元，分别是  $\{(1,3), (2,3), (2,4)\}$ ， $\{(3,5), (4,4), (4,5), (5,5)\}$ ， $\{(5,2), (6,1), (6,2), (6,3), (7,1), (7,2), (7,3)\}$ ， $\{(5,7), (6,7), (7,6), (7,7)\}$ 。在图 6-13b 中，属于同一图元的像素被编上相同的标号。

在识别图元的程序中，采纳了许多在解决电路布线问题时所使用的策略。为了轻松地在图像中移动，在图像周围包上一圈空白像素（即 0 像素）。采用数组 *offset* 来确定与一个给定像素相邻的像素。通过逐行扫描像素来识别图元。当遇到一个没有标号的图元像素时，就给它指定一个图元编号（使用数字 2,3,... 作为图元编号），该像素就成为一个新图元的种子。通过识别和标记与种子相邻的所有图元像素，可以确定图元中的其他像素。我们把与种子相邻的像素称为 1-间距像素。接下来要识别和标记与 1-间距像素相邻的所有无标记图元像素，这些像素被称为

2-间距像素。之后继续识别和标记与2-间距像素相邻的无标记图元像素。这个过程一直持续到再也找不到新的、相邻的无标记图元像素为止。

		1				
		1	1			
				1		
			1	1		
	1			1		1
1	1	1				1
1	1	1			1	1

		2				
		2	2			
				3		
			3	3		
	4			3		5
4	4	4				5
4	4	4			5	5

图6-13 识别图元

a) 7 × 7 图像 b) 标记图元

上述图元识别的过程与确定布线过程中用距离值（与起始方格的距离）来标记网格位置的过程非常类似，因此，识别图元的程序6-10与程序6-9也很类似。

程序6-10首先在图像周围包上一圈背景像素（即0像素），并对数组offset进行初始化。接下来的两个for循环通过扫描图像来寻找下一个图元的种子。种子应是一个无标记的图元像素，对种子来说，有 $\text{pixel}[r][c]=1$ 。将 $\text{pixel}[r][c]$ 从1变成id（图元编号），即可把图元编号设置为种子的标号。接下来借助于链表队列的帮助（也可以使用公式化队列、链表堆栈或公式化堆栈），可以识别出该图元中的其余像素。当函数Label结束时，所有的图元像素都已经获得了一个标号。

初始化“围墙”需耗时 $\Theta(m)$ ，初始化offset需耗时 $\Theta(1)$ 。尽管条件 $\text{pixel}[r][c]==1$ 被检查了 $m^2$ 次，但它为true的次数只有cnum次，其中cnum为图像中图元的总数。对于任一个图元来说，识别并标记该图元的每个像素（种子除外）所需要的时间为 $\Theta(\text{cnum})$ 。由于任意一个像素都不会同时属于两个以上的图元，因此，识别并标记所有非种子图元像素所需要的总时间为 $\Theta(\text{图像中图元像素总数}) = \Theta(\text{输入图像中值为1的像素数目}) = \Theta(m^2)$ 。因此，函数Label总的复杂度为 $\Theta(m^2)$ 。

程序6-10 识别图元

```
void Label()
// 识别图元

// 初始化“围墙”
for (int i = 0; i <= m+1; i++) {
    pixel[0][i] = pixel[m+1][i] = 0; // 底和顶
    pixel[i][0] = pixel[i][m+1] = 0; // 左和右
}

// 初始化offset
Position offset[4];
offset[0].row = 0; offset[0].col = 1; // 右
```



```
offset[1].row = 1; offset[1].col = 0; // 下
offset[2].row = 0; offset[2].col = -1; // 左
offset[3].row = -1; offset[3].col = 0; // 上

int NumOfNbrs = 4; // 一个像素的相邻像素个数
LinkedList<Position> Q;
int id = 1; // 图元id
Position here, nbr;

// 扫描所有像素
for (int r = 1; r <= m; r++) // 图像的第 r 行
    for (int c = 1; c <= m; c++) // 图像的第 c 列
        if (pixel[r][c] == 1) { // 新图元
            pixel[r][c] = ++id; // 得到下一个 id
            here.row = r; here.col = c;

            do { // 寻找其余图元
                for (int i = 0; i < NumOfNbrs; i++) {
                    // 检查当前像素的所有相邻像素
                    nbr.row = here.row + offset[i].row;
                    nbr.col = here.col + offset[i].col;
                    if (pixel[nbr.row][nbr.col] == 1) {
                        pixel[nbr.row][nbr.col] = id; Q.Add(nbr); } // end of if and for

                    // 还有未探索的像素吗?
                    if (Q.IsEmpty()) break;
                    Q.Delete(here); // 一个图元像素
                } while(true);

            } // 结束if 和for
        }
```

#### 6.4.4 工厂仿真

##### 1. 问题描述

一间工厂由 $m$ 台机器组成。工厂中所执行的每项任务都由若干道工序构成，一台机器用来完成一道工序，不同的机器完成不同的工序。一旦一台机器开始处理一道工序，它会连续不断地进行处理，直到该工序被完成为止。

例6-1 一个生产金属片的工厂可能对于如下每道工序都有一台相应的机器：设计、切割、钻孔、挖孔、修边、造型和焊接。每台机器每次处理一道工序。

每项任务都包含若干道工序。例如，为了在一个新房子内安装暖气管道和空调管道，首先需要花一些时间来进行设计，然后根据设计要求把整块的金属片切割成各种尺寸的金属片，在金属片上钻孔或挖孔（取决于孔的大小），把金属片塑造成管道，焊接管缝，并对粗糙的边进行修剪和打磨。

对于一项任务中的每道工序来说，都有两个属性：一是工时（即完成该道工序需要多长时间）

间), 一是执行该工序的机器。一项任务中的各道工序必须按照一定的次序来执行。一项任务的执行是从处理第一道工序的机器开始的, 当第一道工序完成后, 任务转至处理第二道工序的机器, 依此进行下去, 直到最后一道工序完成为止。当一项任务到达一台机器时, 若机器正忙, 则该任务将不得不等待。事实上, 很可能有多项任务同时在一台机器旁等待。

在工厂中每台机器都可以有如下三种状态: 活动、空闲和转换。在活动状态, 机器正在处理一道工序, 而在空闲状态机器无事可做。在转换状态, 机器刚刚完成一道工序, 并在为一项新任务的执行做准备, 比如机器操作员可能需要清理机器并稍作休息等。每台机器在转换状态期间所花费的时间可能各不相同。

当一台机器可以处理一项新任务时, 它可能需要从各个等待者中挑选一项任务来执行。在这里, 每台机器都按照 FIFO 的方式来处理等待者, 因此每台机器旁的等待者构成了一个 FIFO 队列。在其他类型的工厂中, 可以为每项任务指定不同的优先权, 当机器变成空闲时, 从等待者中首先选择具有最高优先权的任务来执行。

一项任务最后一道工序的完成时间被称为任务完成时间。一项任务的长度等于其所有工序的执行时间之和。如果一项长度为  $l$  的任务在 0 时刻到达工厂并在  $f$  时刻结束, 那么它在各机器队列中所花费的等待时间恰好为  $f - l$ 。为了让顾客满意, 希望尽量减少任务在机器队列中的等待时间。如果能够知道每项任务所花费的等待时间是多少, 并且知道哪些机器所导致的等待时间最多, 就可以据此来改进和提高工厂的效能。

在对工厂进行仿真时, 我们只是让任务在机器间流动, 并没有实际执行任何工序, 而是采用一个模拟时钟来进行仿真计时, 每当一道工序完成或一项新任务到达工厂时, 模拟时钟就推进一个单位。在完成老任务时, 将产生新的任务。每当一道工序完成或一项新任务到达工厂时, 我们称发生了一个事件 (event)。另外, 还存在一个启动事件 (start event), 用来启动仿真过程。下面的例子演示了在仿真期间没有新任务到达工厂时的仿真过程。

例6-2 考察一间工厂, 它由  $m=3$  台机器构成, 可以处理  $n=4$  项任务。假定所有四项任务都在 0 时刻出现并且在仿真期间不再有新的任务出现。仿真过程一直持续到所有任务都已完成为止。

三台机器  $M_1$ 、 $M_2$  和  $M_3$  的转换状态所花费的时间分别为 2、0 和 1。因此, 当一道工序完成时, 机器  $M_1$  在启动下一道工序之前必须等待 2 个时间单元, 机器  $M_2$  可以立即启动下一道工序, 机器  $M_3$  必须等待 1 个时间单元。图 6-14a 分别列出了四项任务的特征。例如, 1 号任务有 3 道工序。每道工序用形如 (machine, time) 的值对来描述。1 号任务的第一道工序将在  $M_1$  上完成, 需花费的时间为 2 个时间单元, 第二道工序将在  $M_2$  上完成, 需花费的时间为 4 个时间单元, 第三道工序将在  $M_1$  上完成, 需花费的时间为 1 个时间单元。各项任务的长度分别为 7, 6, 8 和 4。

图 6-14b 列出了工厂仿真的过程。起始时刻, 首先按照各任务的第一道工序把 4 项任务分别放入相应的机器队列中。1 号任务和 3 号任务的第一道工序将在  $M_1$  上执行, 因此这两项任务被放入  $M_1$  的队列中。2 号任务和 4 号任务的第一道工序将在  $M_3$  上执行, 因此这两项任务被放入  $M_3$  的队列中。 $M_2$  的队列为空。在启动仿真过程之初, 所有 3 台机器都是空闲的。符号 I 表示机器处于空闲状态。若一台机器处于空闲状态, 那么该机器完成当前工序 (实际上不存在) 的时间没有定义, 可用符号  $L$  来表示。

仿真从 0 时刻开始, 即第一个事件——启动事件——出现在 0 时刻。此时, 每个机器队列中的第一个任务被调度到相应的机器上执行。因此, 1 号任务的第一道工序被调度到  $M_1$  上执行, 2 号任务的第一道工序被调度到  $M_3$  上执行。这时  $M_1$  的队列中仅包含 3 号任务, 而  $M_3$  的队列中仅

任务	工序数日	工序
1	3	(1,2) (2,4) (1,1)
2	2	(3,4) (1,2)
3	2	(1,4) (2,4)
4	2	(3,1) (2,3)

a)

时间	机器队列			活动的任务			完成时间		
	M1	M2	M3	M1	M2	M3	M1	M2	M3
Init	1,3	—	2,4	I	I	I	L	L	L
0	3	—	4	1	I	2	2	L	4
2	3	—	4	C	1	2	4	6	4
4	2	—	4	3	1	C	8	6	5
5	2	—	—	3	1	4	8	6	6
6	2,1	4	—	3	C	C	8	9	7
7	2,1	4	—	3	C	I	8	9	L
8	2,1	4,3	—	C	C	I	10	9	L
9	2,1	3	—	C	4	I	10	12	L
10	1	3	—	2	4	I	12	12	L
12	1	3	—	C	C	I	14	15	L
14	—	3	—	1	C	I	15	15	L
15	—	—	—	C	3	I	17	16	L
16	—	—	—	C	C	I	19	19	L

b)

图6-14 工厂仿真示例

a) 任务特征 b) 仿真

包含4号任务，M2的队列仍然为空。这样，1号任务成为M1上的活动任务，2号任务成为M3上的活动任务，M2仍然为空闲。M1的结束时间变成2（当前时刻0+工序时间2），M3的结束时间变成4。

下一个事件在时刻2出现，这个时刻是根据最小的机器完成时间来确定的。在时刻2，M1完成了它的当前活动工序，它是1号任务的工序。此后1号任务将被移动到M2上以执行下一道工序。由于M2是空闲的，因此1号任务的第二道工序将立即开始执行，这道工序将在第6个时刻完成（当前时刻2+工时4）。从时刻2开始，M1进入转换状态并将持续2个时间单元，这期间，M1的当前工序被设置为C，其完成时间为时刻4。

在时刻4，M1和M3都完成了它们自己的当前工序。由于M1完成的是一个“转换”工序，所以它开始执行新的任务。为此，从M1的队列中选择第一个任务——3号任务。3号任务第一个工序的长度为4，所以该工序的结束时间为8，8也同时成为M1的完成时间。M3上的2号任务在完成其第一道工序之后需移至M1上继续执行，由于M1正忙，所以2号任务被放入M1的队列，M3则进入转换状态，转换状态的结束时刻为5。

依此类推能够给出后续的事件序列。2号和4号任务在第12时刻完成，1号任务在第15时刻完成，3号任务在第19时刻完成。由于2号任务的长度为6，而它的完成时刻为12，所以2号任务在队列中所花费的等待时间为12-6=6个时间单元。类似地，4号任务在队列中的等待时间为12-4=8个时间单元，1号和3号任务的等待时间分别为8和11个时间单元。总的等待时间为33个时间单元。

可以确定这33个单元等待时间在3台机器上的具体分布。例如，4号任务在0时刻进入M3的队列，直到时刻5才开始被执行，所以该项任务在M3的队列中所花费的等待时间为5个时间单元。其他的任务都不需要在M3上等待，因此在M3上所花费的总的等待时间为5。仔细检查图6-14b，可以计算出在M1和M2上所花费的等待时间分别为18和10个时间单元。正如我们所预料的，各任务的等待时间之和（33）就等于在各机器上所花费的等待时间之和。

## 2. 高级仿真器设计

在设计仿真器时，假定所有的任务都在0时刻出现，并且在仿真过程中不再出现其他新的任务，此外还假定仿真过程将一直持续到所有任务都完成为止。

由于仿真器是一个相当复杂的程序，因此可以把它分解成若干个模块。仿真器所执行的主要任务是：输入数据，把各任务按其第一道工序放入相应队列；执行启动事件（即装入初始任务）；处理所有事件（即执行实际仿真）和输出队列等待时间。分别使用一个C++函数来实现仿真器的每项任务。每个函数都可能引发诸如NoMem和BadInput（非法的输入数据）的异常。主函数见程序6-11，其中的catch语句可用若干条catch语句来替代，每条catch语句用于引发一种异常，并输出不同的信息。练习19要求你完成这项工作。

程序6-11 工厂仿真的主函数

---

```
void main(void)
{
    //工厂仿真
    try {
        InputData(); // 获取机器和任务数据
        StartShop(); // 启动
        Simulate(); // 仿真
        OutputStats(); // 输出机器等待时间
    } catch (...) {
        cout << "An exception has occurred" << endl;
    }
}
```

---

## 3. 类Task

在设计程序6-11中所调用的四个函数之前，必须设计所需要的数据对象，这些数据对象包括工序、任务、机器和事件表。每个工序都由两部分构成：machine（该工序将在这台机器上处理）和time（完成该工序所需要的时间）。程序6-12给出了类Task的定义。由于对机器从1至m编号，所以machine是int类型（也可使用类型unsigned）。假定所有的时间都是整数，time的类型被定义为long以便于执行长时间的仿真。类Task有两个友元：类Job和函数MoveToNextMachine。把Job定义为Task的友元是因为Job需要访问Task的私有成员。也可以避免这种授权，方法是为Task定义一些共享成员函数，用以设置和获取machine和time的值。

## 4. 类Job

每项任务都有一个相关的工序表，每道工序按表中的次序执行。因此，工序表可以被描述成一个队列TaskQ。为了确定一项任务所花费的总共的等待时间，需要知道该任务的长度和完成时间。完成时间由计时来确定，而长度则为各工序的工时之和。为了确定任务的长度，我们为Job定义了一个私有数据成员Length。

程序6-12给出了类Job的定义。TaskQ被定义成一个链表队列，用来保存各道工序。由于在

输入数据时已经知道一项任务所包含的工序数目，因此可以把 TaskQ 定义成一个指针，以便动态构造一个足够大的公式化队列，以容纳所期望数量的工序。对于这种受限制的仿真器来说，这个定义工作得很好，因为我们假定在仿真开始之后不再有新的任务进入工厂。若使用链表队列，则只要完成一道工序，即可释放该工序所占用的队列节点，被释放的节点可重新用来构造新任务的工序队列。若使用公式化队列，则仅当整个任务都已经完成时才释放空间，因此，仿真很可能因为空间不足而失败，即使中只剩下很少量的工序未被完成。

私有成员 ArriveTime 用于记录一项任务进入当前机器队列的时间，可用来确定任务在这个队列中的等待时间。任务标识符存储在 ID 之中，仅在输出任务的总等待时间时，才会使用该标识符。

共享成员函数 AddTask 用于向任务的工序队列中添加一道工序，该工序将在机器 p 上执行，需要 t 个时间单元。该函数仅用于数据输入过程。当从一个机器队列中移出一项任务并将其变成活动状态时，需要使用共享成员函数 DeleteTask。该函数从工序队列（工序队列用于保存尚未被处理的工序）中删除排在队首的工序，然后返回该工序的时间并将其加入所属任务的长度中。当任务的最后一道工序完成时，Length 的值即为该任务的长度。

程序6-12 类Task和Job

```
class Task {
    friend class Job;
    friend bool MoveToNextMachine(Job*);
private:
    long time;
    int machine;
};

class Job {
    friend bool MoveToNextMachine(Job*);
    friend Job* ChangeState(int);
    friend void Simulate();
    friend class Machine;
public:
    Job(long id) {ID = id;
        Length = ArriveTime = 0;}
    void AddTask(int p, long t) {
        Task x;
        x.machine = p;
        x.time = t;
        TaskQ.Add(x);}
    long DeleteTask() { // 删除下一工序
        Task x;
        TaskQ.Delete(x);
        Length += x.time;
        return x.time;}
private:
    LinkedQueue<Task> TaskQ;
    long Length;    // 工序时间
    long ArriveTime; // 到达当前队列的时间
```

```
long ID;    // 任务标识符  
};
```

### 5. 类Machine

每台机器都包含如下三个属性：转换时间、当前任务和等待队列。由于任务是一个相当大的对象（有自己的工序队列和其他数据成员），因此把保存等待任务的队列定义成一个指针队列（每个指针指向一个任务）可以大大提高处理效率，这样每个机器队列的操作处理的是指针（很小的对象）而不是任务。

由于每项任务任意时刻只会在一台机器上，所以所有队列总的空间需求与任务的数目直接相关。不过，任务在各个机器队列中的分布会随着仿真的进行而不断变化。某一时刻出现几个很长的队列是完全可能的，而稍后这些队列可能会变短，其他队列会变长。如果采用公式化队列，则必须把每个机器队列的大小都定义成最大可能的值。（否则必须动态调整队列所在数组的大小），此时，需要存储 $m \times (n-1)$ 个任务指针（ $m$ 是机器的数目， $n$ 是任务的数目）。如果使用链表队列，则只需要存储 $n$ 个任务指针和 $n$ 个链接域。因此，我们使用链表队列。

程序6-13给出了类Machine的定义。私有成员JobQ、ChangeTime、TotalWait、NumTasks和Active分别表示指向等待任务的指针队列、机器的转换时间、在该台机器上已花费的总的等待时间、该机器所完成的工序数目和指向当前活动任务的指针。当机器空闲或处于转换状态时，活动任务的指针为0。

共享成员函数IsEmpty 当且仅当任务队列为空时返回true。共享成员函数AddJob向等待队列中添加一项任务。函数SetChange用于在输入期间设置ChangeTime的值。函数Status用于返回该机器迄今为止总的等待时间及完成的工序数目。

所有机器的完成时间被存储在一个事件表中。为了从一个事件转向下一个事件，必须确定最小的机器完成时间。仿真器还需要一个设置一台机器的完成时间的操作，每当一个新任务被调度到一台机器上运行时就要执行该操作。当一台机器变成空闲时，其完成时间被设置成一个很大的数。

程序6-13 类Machine

```
class Machine {  
    friend Job* ChangeState(int);  
public:  
    Machine() {TotalWait = NumTasks = 0;  
               Active = 0;}  
    bool IsEmpty() {return JobQ.IsEmpty();}  
    void AddJob(Job* x) {JobQ.Add(x);}  
    void SetChange(long w) {ChangeTime = w;}  
    void Stats(long& tw, long& nt)  
        {tw = TotalWait;  
         nt = NumTasks;}  
private:  
    LinkedQueue<Job*> JobQ; // 等待队列  
    long ChangeTime; // 机器转换时间  
    long TotalWait; // 机器总的等待时间  
    long NumTasks; // 所处理的工序数目  
    Job *Active; // 指向当前任务的指针  
};
```



## 6. 类EventList

程序6-14给出了类EventList的定义，它用一个一维数组FinishTime来处理事件，其中FinishTime[p]表示机器p的完成时间。初始化时，所有的机器都是空闲的，它们的完成时间都被设置为BigT。机器数目用变量NumMachines来记录。

函数NextEvent用于返回下一个事件对应的机器p和完成时间t。对于一个有m台机器的工厂，寻找最小的完成时间所需要的时间为 $\Theta(m)$ ，因此函数NextEvent的复杂性为 $\Theta(m)$ 。函数SetFinishTime用来设置一台机器的完成时间，其复杂性为 $\Theta(1)$ 。在第9章中将看到两个数据结构——堆和最左树，也可以用它们来描述事件。如果使用堆或最左树，NextEvent和SetFinishTime的复杂性均变成 $O(\log m)$ 。如果所有任务的工序总数为T，仿真器将分别调用 $\Theta(T)$ 次NextEvent和 $\Theta(T)$ 次SetFinishTime。在程序6-14中，调用NextEvent和SetFinishTime所花费的时间为 $\Theta(Tm)$ ，若使用堆或最左树，所需要的时间为 $O(T\log m)$ 。虽然堆或最左树更复杂，但当m比较大时，它们可以大大加快仿真过程。

程序6-14 类EventList

---

```

class EventList {
public:
    EventList(int m, long BigT);
    ~EventList(){delete [] FinishTime;}
    void NextEvent(int& p, long& t);
    long NextEvent(int p) {return FinishTime[p];}
    void SetFinishTime(int p, long t)
        {FinishTime[p] = t;}
private:
    long *FinishTime; // 完成时间数组
    int NumMachines; // 机器总数
};

EventList::EventList(int m, long BigT)
{//对m台机器的完成时间进行初始化
    if (m < 1) throw BadInitializers();
    FinishTime = new long [m+1];
    NumMachines = m;
    // 所有机器均为空闲
    for (int i = 1; i <= m; i++)
        FinishTime[i] = BigT;
}

void EventList::NextEvent(int& p, long& t)
{// 返回下一个事件所对应的机器和完成时间
    // 寻找具有最小完成时间的机器
    p = 1;
    t = FinishTime[1];
    for (int i = 2; i <= NumMachines; i++)
        if (FinishTime[i] < t) { // i 较早完成
            p = i;
            t = FinishTime[i];
        }
}

```

---



### 7. 全局变量

程序6-11的四个函数中所使用的全局变量见程序6-15。多数全局变量的含义从变量名中就可以看出来。Now用来记录当前的模拟时间，每次发生一个新的事件时，就会修改Now的值。LargeTime用来表示空闲机器的完成时间。

程序6-15 仿真器中所使用的全局变量

---

```
//全局变量
long Now = 0;           //当前时间
int m;                 // 机器数
long n;                // 任务数
long LargeTime = 10000; // 空闲机器的完成时间
EventList *EL;         //指向事件表的指针
Machine *M;            // 机器数组
```

---

### 8. 函数InputData

函数InputData（见程序6-16）首先输入工厂中的机器数和任务数，然后创建初始事件表\*EL（其中每台机器的完成时间均被设置为LargeTime）和机器数组M。接下来输入每台机器的转换时间并依次输入每项任务。对于每项任务，首先输入该任务所包含的工序数目，然后按(machine, time)的形式输入每个工序。任务的第一道工序所对应的机器记录在变量p中。当一个任务的所有工序都已输入完毕时，该任务（实际上是指向该任务的指针）被放入机器p的队列中。

程序6-16 输入工厂数据

---

```
void InputData()
{
    // 输入工厂数据
    cout << "Enter number of machines and jobs" << endl;
    cin >> m >> n;
    if (m < 1 || n < 1) throw BadInput();

    // 创建事件表和机器队列
    EL = new EventList(m, LargeTime);
    M = new Machine [m+1];

    // 输入机器等待时间
    cout << "Enter change-over times for machines" << endl;
    for (int j = 1; j <= m; j++) {
        long ct; // 转换时间
        cin >> ct;
        if (ct < 0) throw BadInput();
        M[j].SetChange(ct);
    }

    // 输入n个任务
    Job *J;
    for (int i = 1; i <= n; i++) {
        cout << "Enter number of tasks for job " << i << endl;
        int tasks; // 工序数目
```

```

int first; // 任务的第一道工序所对应的机器
cin >> tasks;
if (tasks < 1) throw BadInput();
J = new Job(i);
cout << "Enter the tasks (machine, time)" << " in process order" << endl;
for (int j = 1; j <= tasks; j++) { // 输入工序
    int p; // 机器数目
    long tt; // 工时
    cin >> p >> tt;
    if (p < 1 || p > m || tt < 1) throw BadInput();
    if (j == 1) first = p; // 任务的第一台机器
    J->AddTask(p,tt); // 添加到工序队列
}
M[first].AddJob(J); // 把任务添加到第一道工序所对应的机器
}
}

```

### 9. 函数StartShop和ChangeState

为了启动仿真，需要从每个机器任务队列中取出第一个任务并放到该机器上执行。由于每台机器的初始状态为空闲状态，为了加载初始任务，需要把机器从空闲状态变成活动状态（在仿真进行过程中也如此）。函数ChangeState(i)可用来转换机器i的状态。在程序6-17中，为了启动仿真，只需对每台机器调用ChangeState即可。

程序6-17 启动仿真

```

void StartShop()
{ // 加载每台机器上的第一项任务
    for (int p = 1; p <= m; p++)
        ChangeState(p);
}

```

程序6-18给出了函数ChangeState的代码。如果机器p空闲或处于转换状态，则ChangeState返回0，否则返回指向当前正在处理的任务的指针。此外，ChangeState(p)可改变机器p的状态。如果机器p本来为空闲或处于转换状态，则让p处理其等待队列中的下一项任务，如果队列为空，则将机器的状态设置为空闲。如果机器p正处理完一项任务，则使其进入转换状态。

如果M[p].Active为0，则机器p要么空闲，要么处于转换状态，LastJob所返回的任务指针为0。如果机器的任务队列为空，则将该机器变成空闲状态，并将其完成时间设置为LargeTime。如果任务队列不为空，则删除队列中的第一个任务并将其变成该机器的活动任务，该任务在队列中所花费的等待时间被累加到该机器的总等待时间之中，同时将该机器所处理的工序数目增加1。接下来从任务的工序队列中删除将要处理的工序，并将机器的完成时间设置为新工序完成的时间。

如果M[p].Active不为0，则表明机器p正忙，需返回指向当前任务的指针，为此将该指针存储在LastJob中。机器的状态将变成转换状态，并持续ChangeTime个时间单元。

程序6-18 修改机器状态

```

Job* ChangeState(int p)

```

```

{// 机器p上的工序已完成，调度下一个任务
Job* LastJob;
if (!M[p].Active) {// 空闲或转换状态
    LastJob = 0;
    // 结束等待，准备处理下一项任务
    if (M[p].JobQ.IsEmpty()) // 没有处于等待状态的任务
        EL->SetFinishTime(p, LargeTime);
    else {// 取出任务Q并处理之
        M[p].JobQ.Delete(M[p].Active);
        M[p].TotalWait +=
            Now - M[p].Active->ArriveTime;
        M[p].NumTasks++;
        long t = M[p].Active->DeleteTask();
        EL->SetFinishTime(p, Now + t);}
    }
else {// M[p]上的工序刚刚完成
    // 进入转换状态
    LastJob = M[p].Active;
    M[p].Active = 0;
    EL->SetFinishTime(p, Now + M[p].ChangeTime);}
return LastJob;
}

```

#### 10. 函数Simulate和MoveToNextMachine

程序6-19中，函数Simulate对所有的事件进行循环，直到最后一项任务结束。 $n$ 是尚未完成的任务数目，因此，当 $n=0$ 时，while循环将结束。在while的每次循环过程中，需确定下一个事件的到达时间并将该时间存入Now。对于产生事件的机器 $p$ ，需改变其状态，如果该机器刚刚处理完一道工序( $J$ 不为0)，则调度任务 $*J$ 并处理该任务的下一道工序。函数MoveToNextMachine可用来对任务进行调度。如果任务 $*J$ 中不再有未处理的工序，则表明该任务已完成，函数MoveToNextMachine将返回false，同时将 $n$ 减1。

函数MoveToNextMachine（见程序6-20）首先检查任务 $*J$ 中是否有尚未被处理的工序。如果没有，则表明该任务已完成，输出该任务的完成时间和等待时间。函数返回false意味着任务 $*J$ 不必再移动到下一台机器上去处理。

若任务 $*J$ 尚未完成，则需确定该任务的下一站——机器 $p$ ，同时将 $*J$ 加入 $p$ 的等待队列之中。如果机器 $p$ 为空闲，则调用函数ChangeState来改变其状态。

程序6-19 处理所有任务

```

void Simulate()
{//处理完所有n项任务
    int p;
    long t;
    while (n) {// 至少剩下一个任务
        EL->NextEvent(p,t); // 下一个完工的机器
        Now = t; // 当前时间
        // 修改机器 p 的状态
        Job *J = ChangeState(p);
    }
}

```

```
//把任务J 移至下一台机器
// 如果 J 已完成，则将 n减1
if (J && !MoveToNextMachine(J)) n--;
}
}
```

程序6-20 把一项任务移至下一道工序对应的机器

```
bool MoveToNextMachine(Job *J)
{
    // 把任务J 移至下一道工序所对应的机器
    // 如果所有工序都已处理完毕，则返回 false
    if (J->TaskQ.IsEmpty()) { // 没有未处理的工序
        cout << "Job " << J->ID << " has completed at " << Now << " Total wait was " << (Now-J->Length)
            << endl;
        return false;
    }
    else { // 有未处理的工序
        // 获取下一道工序所对应的机器
        int p = J->TaskQ.First().machine;
        // 放入p 的等待队列
        M[p].AddJob(J);
        J->ArriveTime = Now;
        // 如果 p 空闲，则立即执行该工序
        if (EL->NextEvent(p) == LargeTime) {
            // 机器为空闲
            ChangeState(p);
        }
        return true;
    }
}
```

## 11. 函数OutputStats

由于一个任务的完成时间和等待时间已由函数MoveToNextMachine输出，因此函数OutputStats只需输出完成所有任务所需时间（它也是最后一个任务的完成时间，在 MoveToNextMachine中已被输出过一次）以及每台机器的统计信息（该机器总的等待时间以及所处理的工序数目）。程序6-21给出了相应的代码。

程序6-21 输出每台机器的等待时间

```
void OutputStats()
{
    // 输出机器的等待时间
    cout << "Finish time = " << Now << endl;
    long TotalWait, NumTasks;
    for (int p = 1; p <= m; p++) {
        M[p].Stats(TotalWait, NumTasks);
        cout << "Machine " << p << " completed " << NumTasks << " tasks" << endl;
        cout << "The total wait time was " << TotalWait;
        cout << endl << endl;
    }
}
```

## 练习

11. 采用 $k$ 个缓冲铁轨,对于所有可能的车厢排列,程序6-7都能成功地进行车厢重排吗?试证明你的结论。

12. 重写程序6-7,假定任意时刻缓冲铁轨 $i$ 中最多只能有 $s_i$ 节车厢。具有最小 $s_i$ 的铁轨作为直接通道。

13. 设计一个完整的用于寻找电路布线的C++程序。程序中应包含如下函数: Welcome函数(用来显示程序的名称和功能);输入函数,用于输入网格的大小、封锁的和空白的网格位置、电线的端点; FindPath函数(见程序6-9);输出函数,用于输出网格及路径。使用书中的例子测试程序的正确性。

14. 设计一个完整的用于进行图元识别的C++程序。程序中应包含如下函数: Welcome函数(用来显示程序的名称和功能);输入函数,输入图像的大小及单色图像数据;函数 Label(见程序6-10);输出函数,用于输出识别后的图像,不同图元中的像素用不同的颜色来表示。使用书例中的图像测试程序的正确性。

15. 采用公式化队列重写函数 Label(见程序6-10)。使用公式化队列来代替链表队列,会有哪些优点和缺点?

16. 采用堆栈来重写函数Label(见程序6-10)。使用堆栈来代替队列,会有哪些优点和缺点?

17. 能否把程序5-5中的堆栈换成队列?为什么?

18. 能否把程序5-12中的堆栈换成队列?为什么?

19. 重写程序6-11,采用多个catch语句分别输出不同的错误信息。每个catch语句应对应一种执行期间可能出现的异常。

\*20. 设计一个增强型的工厂仿真器,允许为任务指定各个相邻工序之间最少的等待时间。在完成一项任务的每一道工序(包括最后一道工序)时,仿真器必须将该任务变成等待状态。因此,当一项任务的一道工序完成时,该任务被立即放入下一个队列,此时,该任务进入等待状态。当一台机器准备启动一道新工序时,它必须跳过队列前部仍处于等待状态的任务。可以把被跳过的任务移动到队列的尾部。

\*21. 设计一个增强型的工厂仿真器,允许在仿真期间有新的任务进入。仿真过程在预先指定的时刻结束,尚未完成的任务均处于未完成状态。

## 6.5 参考及推荐读物

6.4.2节中的电路布线算法选自N.Sherwani. *Algorithms for VLSI Physical Design Automation* 第2版. Kluwer Academic, 1995。书中详细讨论了该算法,并给出了其他算法。

China-pub.com

下载

## 第7章 跳表和散列

对于一个有 $n$ 个元素的有序数组，用折半搜索法进行搜索所需要的时间为 $O(\log n)$ ，而对一个有序链表进行搜索所需要的时间为 $O(n)$ 。我们可以通过对有序链表上的全部或部分节点增加额外的指针，来提高搜索性能。在搜索时，可以通过这些指针来跳过链表中若干个节点，因此没有必要从左到右搜索链表中的所有节点。

增加了向前指针的链表叫作跳表。跳表不仅能提高搜索性能，同时也可以提高插入和删除操作的性能。它采用随机技术决定链表中哪些节点应增加向前指针以及在该节点中应增加多少个指针。采用这种随机技术，跳表中的搜索、插入、删除操作的时间均为 $O(\log n)$ ，然而，最坏情况下时间复杂性却变成 $\Theta(n)$ 。相比之下，在一个有序数组或链表中进行插入/删除操作的时间为 $O(n)$ ，最坏情况下为 $\Theta(n)$ 。

散列法是用来搜索、插入、删除记录的另一种随机方法。与跳表相比，它的插入/删除操作时间提高到 $\Theta(1)$ ，但最坏情况下仍为 $\Theta(n)$ 。尽管如此，在经常将所有元素按序输出或按序号搜索元素时（如寻找第10个最小元素），跳表的执行效率将优于散列。

本章所给出的应用是关于文本压缩和解压缩的应用，该应用用散列实现。所设计的程序基于当前流行的Liv\_Zempel\_Welch算法（简称LZW算法）。

### 7.1 字典

字典（dictionary）是一些元素的集合。每个元素有一个称作key的域，不同元素的key各不相同。有关字典的操作有：

- 插入具有给定关键字值的元素。
- 在字典中寻找具有给定关键字值的元素。
- 删除具有给定关键字值的元素。

抽象数据类型Dictionary的描述见ADT 7-1。若仅按照一个字典元素本身的关键字来访问该元素，则称为随机访问（random access）。而顺序访问（sequential access）是指按照关键字的递增顺序逐个访问字典中的元素。顺序访问需借助于Begin（用来返回关键字最小的元素）和Next（用来返回下一个元素）等操作来实现。对于本章中所实现的部分字典，既可以采用随机访问方式，也可以采用顺序访问方式。

ADT7-1 字典的抽象数据类型描述

---

抽象数据类型Dictionary {

实例

具有不同关键字的元素集合

操作

Create(): 创建一个空字典

Search( $k, x$ ): 搜索关键字为 $k$ 的元素，结果放入 $x$ ；

如果没找到，则返回false，否则返回true

Insert( $x$ ): 向字典中插入元素 $x$



*Delete* ( $k, x$ ) : 删除关键字为  $k$  的元素, 并将其放入  $x$

}

有重复元素的字典 (dictionary with duplicates) 与上面定义的字典相似, 只是它允许许多个元素有相同的关键字。在有重复元素的字典中, 在进行搜索和删除时需要一个规则来消除歧义。也就是说, 如果要搜索(或删除)关键字为  $k$  的元素, 那么在所有关键字为  $k$  值的元素中应该返回(或删除)哪一个呢? 在有些字典应用中, 可能需要: 删除在某个时间以后插入的所有元素。

例7-1 一个班中注册学习数据结构课程的学生构成了一个字典。当有一个新学生注册时, 就要在字典中插入与该学生相关的元素(记录)。当有人要放弃这门课程时, 则删除他的记录。在上课过程中, 老师可以查询字典以得到与某特定学生相关的记录或修改记录(例如, 加入或修改考试成绩)。学生的姓名域可作为关键字。

例7-2 在编译器中定义用户描述符的符号表 (symbol table) 就是一个有重复元素的字典。当定义一个描述符时, 要建立一个记录并插入到符号表中。记录中包括作为关键字的描述符以及其他信息, 如描述符类型(int, float 等) 和(相关的)存储其值的内存地址。因为同样的描述符名可以定义多次(在不同的程序块中), 所以符号表中必然存在有多个记录具有相同的关键字, 搜索结果应是最新插入的元素。只有在程序块的结尾才能进行删除, 所有在开始插入的元素最终都要被删除掉。

## 7.2 线性表描述

字典可以保存在线性序列( $e_1, e_2, \dots$ ) 中, 其中  $e_i$  是字典中的元素, 其关键字从左到右依次增大。为了适应这种描述方式, 可以定义两个类 SortedList 和 SortedChain。前者采用公式化描述的线性表, 如 LinearList 类(见程序3-1), 而后者则采用链表描述, 如 Chain 类(见程序3-8)。

练习1要求设计 SortedList 类。应注意到在 SortedList 中可以采用折半搜索法搜索元素, 因此对有  $n$  个元素的字典进行搜索的时间为  $O(\log n)$ 。进行插入时, 必须确信该字典中没有相同关键字的元素, 这要通过搜索来实现。然后进行插入, 此时要为新元素腾出空间而移动表中  $O(n)$  个元素, 故插入操作的时间是  $O(n)$ 。删除则首先要找到欲删除的元素, 然后再进行删除。在进行搜索之后, 还要移动  $O(n)$  个元素以填补所删除元素的空间, 因此删除的时间复杂性为  $O(n)$ 。

程序7-1、程序7-2和程序7-3给出了类 SortedChain 的定义。E 表示链表元素的数据类型, K 是链表中排序所用到的关键字。类 SortedChainNode 与类 ChainNode(见程序3-8)一样, 只有两个私有成员 data 和 link。类 SortedChain 是类 SortedChainNode 的友元。

程序7-1 类 SortedChain

```
template<class E, class K>
class SortedChain{
public:
    SortedChain() {first = 0;}
    ~SortedChain();
    bool IsEmpty() const {return first == 0;}
    int Length() const;
    bool Search(const K& k, E& e) const;
    SortedChain<E, K>& Delete(const K& k, E& e);
```

```

SortedChain<E ,K>& Insert(const E& e);
SortedChain<E ,K>& DistinctInsert(const E& e);
private:
SortedChainNode<E ,K> *first;
};

```

程序7-2 SortedChain类的成员函数search和delete

```

template<class E, class K>
bool SortedChain<E,K>::Search(const K& k, E& e) const
{// 搜索与k匹配的元素，结果放入 e
// 如果没有匹配的元素，则返回 false

SortedChainNode<E,K> *p = first;

// 搜索与k相匹配的元素
for (; p && p->data < k;
      p = p->link);

// 验证是否与k匹配
if (p && p->data == k) // 与k相匹配
    {e = p->data; return true;}
return false; // 不存在相匹配的元素
}

template<class E, class K>
SortedChain<E,K>& SortedChain<E,K>
::Delete(const K& k, E& e)
{// 删除与k相匹配的元素
// 并将被删除的元素放入 e
// 如果不存在匹配元素，则引发异常 BadInput

SortedChainNode<E,K> *p = first,
                    *tp = 0; //跟踪p

// 搜索与k相匹配的元素
for (; p && p->data < k; tp = p; p = p->link;
      )

// 验证是否与k匹配
if (p && p->data == k) {找到一个相匹配的元素
    e = p->data; // 保存data域

// 从链表中删除p所指向的元素
if (tp) tp->link = p->link;
else first = p->link; // p是链首节点

delete p;

```

```

        return *this;}
    throw BadInput(); // 不存在相匹配的元素
}

```

程序7-3 有序链表中的插入操作

```

template<class E, class K>
SortedChain<E,K>& SortedChain<E,K>::DistinctInsert(const E& e)
{// 如果表中不存在关键值与 e 相同的元素，则插入 e
 //否则引发异常 BadInput

    SortedChainNode<E,K> *p = first, *tp = 0; // 跟踪 p

    // 移动 tp 以便把 e 插入到 tp 之后
    for (; p && p->data < e; tp = p, p = p->link);

    // 检查关键值是否重复
    if (p && p->data == e) throw BadInput();

    // 若没有出现重复关键值，则产生一个关键值为 e 的新节点
    SortedChainNode<E,K> *q = new SortedChainNode<E,K>;
    q->data = e;

    // 将新节点插入到 tp 之后
    q->link = p;
    if (tp) tp->link = q;
    else first = q;

    return *this;
}

```

类SortedChain提供了两种插入操作，DistinctInsert操作保证链中所有元素有不同的关键字，而Insert允许有相同的关键字。虽然字典应用要求DistinctInsert操作，但有序链表的其他应用可能会用到Insert操作。

析构函数、Length、Output 和<<的重载的代码与类Chain（见程序3-8）一样，在这里不再重复。可以通过删除DistinctInsert中搜索重复元素的那段代码（即程序7-3中的第一条if语句）来得到Insert。从代码中可以看出，在有n个节点的链表中，搜索、插入、删除的时间均为 $O(n)$ 。

在类SortedChain中，需定义有关数据类型E的操作符<<，==，!=和<，至于用户自定义的数据类型，可以采用C++的操作符重载机制来定义各种操作符。许多应用要求只根据E的关键字来实现这些操作。实现重载操作的一个简单方法是重载类型转换符以实现类型E到类型K的转换（K的类型为key）。例如，当K是long时，可以在E的类定义中加入如下语句：

```
operator long() const {return key;}
```

可以扩充所有的SortedList和SortedChain以提供高效的顺序访问。在这两种情况下Begin和

Next返回一个元素所需要的时间均为  $\Theta(1)$ 。

## 练习

1. 采用公式化描述方法定义 C++ 类 SortedList。要求提供与 SortedChain 中同样的成员函数。编写所有函数的代码，并用合适的数据测试代码。

2. 扩充 SortedChain 类，使其包括顺序访问函数 Begin 和 Next。Begin 用来返回字典中第一个元素，Next 用来返回字典中下一个元素（采用从小到大的排列顺序）。在没有第一个或下一个元素的情况下，这两个函数均返回 0。两个函数的复杂度均应为  $\Theta(1)$ 。测试代码的正确性。

3. 修改类 SortedChain，使用一条具有头节点和尾节点的链表。尾节点用来存放比其他元素值都大的元素。采用尾节点可以简化代码。

## 7.3 跳表描述

### 7.3.1 理想情况

在一个使用有序链表描述的具有  $n$  个元素的字典中进行搜索，至多需进行  $n$  次比较。如果在链中部节点加一个指针，则比较次数可以减少到  $n/2+1$ 。搜索时，首先将欲搜索元素与中间元素进行比较。如果欲搜索的元素较小，则仅需搜索链表的左半部分，否则，只要在链表右半部分进行比较即可。

例7-3 图7-1a 的有序链表中有七个元素。该链表有一个头节点和一个尾节点。节点中的数是该节点的值。对该链表的搜索可能要进行七次比较。可以采用图 7-1b 中的办法，把最坏情况下的比较次数减少为四次。搜索一个元素时，首先将它与中间元素进行比较，然后根据得到的结果，或者与链的左半部比较或者与右半部比较。例如如果要找值为 26 的元素，只需查找 40 左边的元素。如果要查找值为 75 的元素，那么只对 40 以后的元素进行搜索即可。

也可以象图 7-1c 中一样，分别在链表左半部分和右半部分的中间节点再增加一个指针，以便进一步减少最坏情况下的搜索比较次数。在该图中有三条链。0 级链就是图 7-1a 中的初始链，包括了所有七个元素。一级链包括第二，四，六个元素，而 2 级链只包括第四个元素。为了寻找值为 30 的元素，首先与中间元素比较。在 2 级链中寻找元素所需要的时间为  $\Theta(1)$ 。由于  $30 > 40$ ，因此要搜索该链左半部分的中间元素。采用 1 级链进行搜索所花费的时间为  $\Theta(1)$ 。又因为  $30 > 24$ ，故需在 0 级链中继续进行查找，把该元素与链中下一个元素进行比较。

考察另一个例子。设要查找的元素值为 77。首先与 40 比较， $70 > 40$ ，则在 1 级链中与 75 比较。由于  $77 > 75$ ，因此在 0 级链中与 75 后面的 80 比较。这时可以得知 77 不在此字典中。采用图 7-1c 中的 3 级链结构，对所有的搜索至多需三次比较。3 级链结构允许在有序链表中进行折半搜索。

通常 0 级链包括  $n$  个元素，1 级链包括  $n/2$  个元素，2 级链包括  $n/4$  个元素，而每  $2^i$  个元素有一个  $i$  级链指针。当且仅当一个元素在  $0 \sim i$  级链上，但不在  $i+1$  级（若该链存在）链上时，我们就说该元素是  $i$  级链元素。在图 7-1c 中，40 是 2 级链上唯一的元素而 75 是 1 级链元素。20、30、60、80 是 0 级链元素。

图 7-1c 所示的结构为跳表（skip list）。在该结构中有一组有层次的链。0 级链是包含所有

元素的有序链表，1级链是0级链的一个子集。 $i$ 级链所包含的元素是 $i-1$ 级链的子集。图7-1c中， $i$ 级链上所有的元素均在 $i-1$ 级链上。

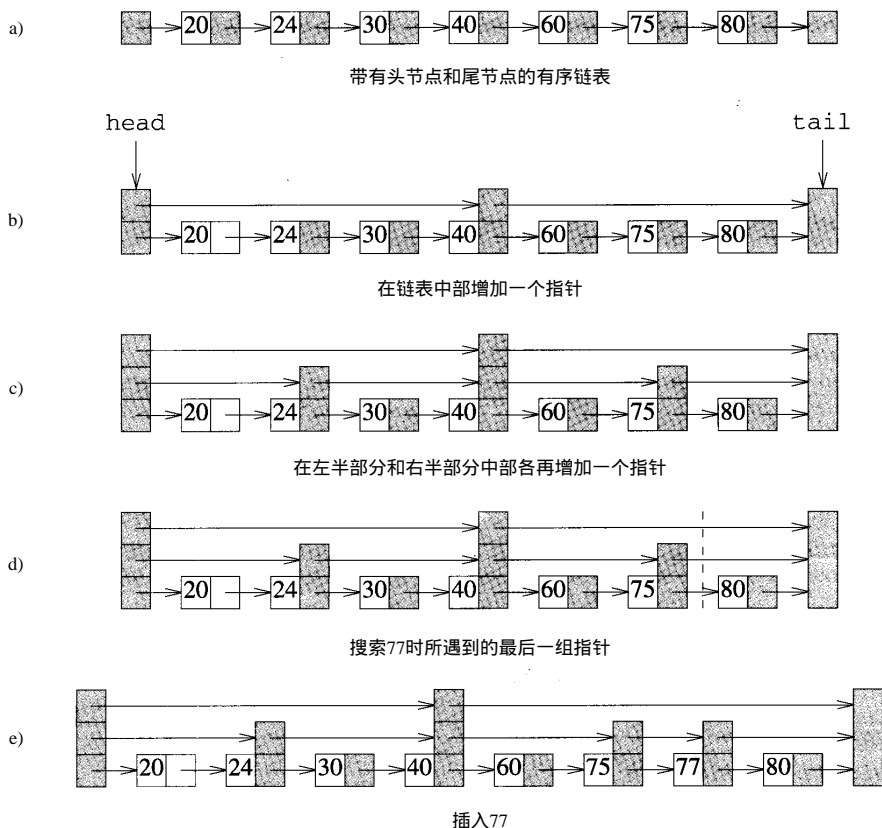


图7-1 有序链表的快速搜索

### 7.3.2 插入和删除

在进行插入和删除时，要想保持图7-1c的跳表结构，必须耗时 $O(n)$ 。注意到在这种结构中，有 $n/2^i$ 个元素为 $i$ 级链元素，所以在进行插入时应尽量逼近这种结构。在进行插入时，新元素属于 $i$ 级链的概率为 $1/2^i$ 。在确定新元素的级时，应考虑各种可能的情况。因此，把新元素作为 $i$ 级链元素的可能性为 $p^i$ 。图7-1c中 $p=0.5$ 。对于一般的 $p$ ，链的级数为 $(\log_{1/p} n) + 1$ 。在这种情况下，每 $p$ 个 $i-1$ 级链中有一个在 $i$ 级链中。

假设要插入的元素为77，首先要通过搜索以确定链中没有此元素。在搜索中，最后一个2级链指针存储在40的指针域中，而最后一个1级链指针存储在75的指针域中。在图7-1d中，这几条指针用虚线标出。新元素插在75和80之间，如图7-1d中的虚线所示。插入时，要为新元素分配一个级，分配过程由随机数产生器完成随机数产生器将在后面介绍。

若新元素为 $i$ 级链元素，则仅影响由虚线断开的 $0 \sim i$ 级链指针。图7-1e给出新插入元素77作为1级链元素时链表的结构。

对于删除操作，我们无法控制其结构。要删除图7-7e中的元素77，首先要找到77。最后所遇到的链指针是节点40中的2级链指针、节点75中的1级链指针和0级链指针。在这些链指针中，因为77为1级链元素，所以只需改变0级和1级链指针即可。当这些指针变成指向77后面的元素

时,就得到图7-1d 的结构。

### 7.3.3 级的分配

在级基本的分配过程中,可以观察到,在一般跳表结构中, $i-1$ 级链中的元素属于 $i$ 级链的概率为 $p$ 。假设有一随机数产生器所产生的数在0到RAND\_MAX间。则下一次所产生的随机数小于等于 $\text{CutOff} = p * \text{RAND\_MAX}$ 的概率为 $p$ 。因此,若下一随机数小于等于 $\text{CutOff}$ ,则新元素应在1级链上。现在继续确定新元素是否在2级链上,这由下一个随机数来决定。若新的随机数小于等于 $\text{CutOff}$ ,则该元素也属于2级链。重复这个过程,直到得到一随机数大于 $\text{CutOff}$ 为止。

故可以用下面的代码为要插入的元素分配级。

```
int lev = 0;
while (rand() <= CutOff) lev++;
```

这种方法潜在的缺点是可能为某些元素分配特别大的级,从而导致一些元素的级远远超过  $\log_{1/p} N$ , 其中 $N$ 为字典中预期的最大数目。为避免这种情况,可以设定一个上限 lev。在有 $N$ 个元素的跳表中,级MaxLevel的最大值为

$$\lceil \log_{1/p} N \rceil - 1 \quad (7-1)$$

可以采用此值作为上限。

另一个缺点是即使采用上面所给出的上限,但还可能存在下面的情况,如在插入一个新元素前有三条链,而在插入之后就有了10条链。这时,新插入元素的为9级,尽管在前面插入中没有出现3到8级的元素。也就是说,在此插入前并未插入3,4,...,8级元素。既然这些空级没有直接的好处,那么可以把新元素的级调整为3。

例7-4 用跳表表示一个最多有1024个元素的字典。设 $p=0.5$ ,则MaxLevel为 $\log_2 1024 - 1 = 9$ 。若随机数产生器的 $\text{RAND\_MAX} = 2^{32} - 1$ ,则 $\text{CutOff} = 2^{31} - 1$ 。新产生的随机数小于等于 $\text{CutOff}$ 的概率为0.5。

假定我们首先从一个空字典开始,该字典用具有头节点和尾节点的跳表结构描述,头节点有10个指针,每个指针对应一条链表,且从头节点指向尾节点。

当插入第一个元素时,为其在0到9(MaxLevel)之间分配一个级。若所分配的级为9,则在插入第一个元素时,要修改九个指针。另一方面,由于没有0,1,...,8级元素,故可以把该元素的级改为0,这样只需修改一条指针即可。

另一种级的分配方法是把随机数产生器的值分为几段。第一段包括范围的 $1 - 1/p$ ,第二段包括 $1/p - 1/p^2$ 等等。若产生的随机数不在第 $i$ 段中,则为此元素分配 $i-1$ 级。

### 7.3.4 类SkipNode

跳表结构的头节点需有足够的指针域,以满足可能构造最大级数的需要,而尾节点不需要指针域。每个存有元素的节点都有一个data域和(级数+1)个指针域。在程序7-4中可以遇到所有类型的节点。指针域由数组link表示,其中link[i]表示 $i$ 级链指针。构造函数为指针数组分配空间。对于一个lev级链元素,其size值应为lev+1。

程序7-4 类SkipNode

```
template<class E, class K>
class SkipNode {
```

```
friend SkipList<E,K>;
private:
    SkipNode(int size)
    {link = new SkipNode<E,K> *[size];}
    ~SkipNode() {delete [] link;}
    E data;
    SkipNode<E,K> **link; // 一维指针数组
};
```

### 7.3.5 类SkipList

程序7-5给出了类SkipList的定义。MaxE是字典的最大容量。虽然在给出的代码中允许元素数目超过MaxE，但若元素数目不超过MaxE，平均性能会更好一些。一个元素既在i-1级链上又在i级链上的概率为p，Large是一个比字典中任意一个数均大的值。尾节点的值Large。0级链上的值（不包括头节点，因其没有值）从左到右按升序排列。

程序7-5 类SkipList

```
template<class E, class K>
class SkipList {
public:
    SkipList(K Large, int MaxE = 10000, float p = 0.5);
    ~SkipList();
    bool Search(const K& k, E& e) const;
    SkipList<E,K>& Insert(const E& e);
    SkipList<E,K>& Delete(const K& k, E& e);
private:
    int Level();
    SkipNode<E,K> *SaveSearch(const K& k);
    int MaxLevel; // 所允许的最大级数
    int Levels; // 当前非空链的个数
    int CutOff; // 用于确定级号
    K TailKey; // 一个很大的key值
    SkipNode<E,K> *head; // 头节点指针
    SkipNode<E,K> *tail; // 尾节点指针
    SkipNode<E,K> **last; // 指针数组
};
```

程序7-6给出了构造函数和析构函数。构造函数初始化CutOff、Levels（当前出现的最大级数）、MaxLevel、TailKey（所有元素值均小于此值）和用来为新元素分配级的随机数产生器。构造函数同时也为头节点和尾节点分配空间。在插入和删除操作之前进行搜索时，所遇到的每条链上的最后一个元素均被放入数组last中。头节点中MaxLevel+1个用于指向各级链的指针被初始化为指向尾节点。构造函数的时间复杂性为 $\Theta(\text{MaxLevel})$ 。

程序7-6 构造函数和析构函数

```
template<class E, class K>
SkipList<E,K>::SkipList(K Large, int MaxE, float p)
```



```

{ // 构造函数
    CutOff = p * RAND_MAX;
    MaxLevel = ceil(log(MaxE) / log(1/p)) - 1;
    TailKey = Large;
    randomize(); // 初始化随机发生器
    Levels = 0; // 对级号进行初始化

    // 创建头节点、尾节点以及数组 last
    head = new SkipNode<E,K> (MaxLevel+1);
    tail = new SkipNode<E,K> (0);
    last = new SkipNode<E,K> *[MaxLevel+1];
    tail->data = Large;

    // 将所有级均置空，即将 head 指向 tail
    for (int i = 0; i <= MaxLevel; i++)
        head->link[i] = tail;
}

template<class E, class K>
SkipList<E,K>::~SkipList()
{ // 删除所有节点以及数组 last
    SkipNode<E,K> *next;

    // 通过删除 0 级链来删除所有节点
    while (head != tail) {
        next = head->link[0];
        delete head;
        head = next;
    }
    delete tail;

    delete [] last;
}

```

析构造函数释放链表中用到的所有空间。其复杂性为  $O(n)$  ( $n$  为 0 级链的长度)。搜索、插入和删除函数均要求对  $E$  进行重载，以便在  $E$  的成员之间、 $E$  与  $K$  的成员之间进行比较。从  $K$  到  $E$  的赋值和转换也必须定义。当每个元素都有一个整数域  $data$  和一个长整数域  $key$ ，且元素的值由  $key$  给出时，可使用程序 7-7 所定义的重载。

程序 7-7 跳表的操作符重载

```

class element {
    friend void main(void);
public:
    operator long() const {return key;}
    element& operator =(long y)
    {key = y; return *this;}
private:
    int data;
    long key;
};

```

SkipList类有两个搜索函数(见程序7-8)。当需要定位一个值为  $k$  的元素时, 可用共享成员函数 `Search`。若找到要搜索的元素, 则将该元素返回到 `e` 中, 并返回 `true`, 否则返回 `false`。`Search`从最高级链( `Levels` 级, 仅含一个元素) 开始查找, 一直到 0级链。在每一级链中尽可能地逼近要查找的元素。当从 `for`循环退出时, 正好处在欲寻找元素的左边。与 0级链中的下一个元素进行比较, 即可确定要找的元素是否在跳表中。

第二个搜索函数为私有成员函数 `SaveSearch`, 由插入和删除操作来调用。`SaveSearch`不仅包含了 `Search`的功能, 而且可把每一级中遇到的最后一个节点存放在数组 `last`之中。

程序7-8 在跳表中搜索

---

```
template<class E, class K>
bool SkipList<E,K>::Search(const K& k, E& e) const
{// 搜索与k相匹配的元素, 并将所找到的元素放入 e
// 如果不存在这样的元素, 则返回 false
if (k >= TailKey) return false;

// 调整指针p, 使其恰好指向可能与k匹配的节点的前一个节点
SkipNode<E,K> *p = head;
for (int i = Levels; i >= 0; i--) // 逐级向下
    while (p->link[i]->data < k) // 在第i级链中搜索
        p = p->link[i]; // 指针

// 检查是否下一个节点拥有关键值 k
e = p->link[0]->data;
return (e == k);
}

template<class E, class K>
SkipNode<E,K> * SkipList<E,K>::SaveSearch(const K& k)
{// 搜索 k 并保存最终所得到的位置
// 在每一级链中搜索
// 调整指针p, 使其恰好指向可能与k匹配的节点的前一个节点
SkipNode<E,K> *p = head;
for (int i = Levels; i >= 0; i--) {
    while (p->link[i]->data < k)
        p = p->link[i];
    last[i] = p;
}
return (p->link[0]);
}
```

---

程序7-9给出了在跳表中插入元素并为其分配级的代码。若欲插入元素的值不比 `TailKey`小, 或表中已有与该值相同的元素, `Insert`将引发 `BadInput`异常。若没有足够的空间进行插入, 则由 `new`引发一个 `NoMem`异常。当元素 `e` 被成功插入后, `Insert` 将返回跳表。

程序7-9 向跳表中插入元素

---

```
template<class E, class K>
int SkipList<E,K>::Level()
```

---

```

{ // 产生一个随机级号，该级号 <= MaxLevel
  int lev = 0;
  while (rand() <= CutOff)
    lev++;
  return (lev <= MaxLevel) ? lev : MaxLevel;
}

template<class E, class K>
SkipList<E,K>& SkipList<E,K>::Insert(const E& e)
{ // 如果不存在重复，则插入 e
  K k = e; // 抽取关键值
  if (k >= TailKey) throw BadInput(); // 关键值太大

  // 检查是否重复
  SkipNode<E,K> *p = SaveSearch(k);
  if (p->data == e) throw BadInput(); // 重复

  // 不重复，为新节点确定级号
  int lev = Level(); // 新节点的级号
  // fix lev to be <= Levels + 1
  if (lev > Levels) {lev = ++Levels; last[lev] = head;}

  // 产生新节点，并将新节点插入 p 之后
  SkipNode<E,K> *y = new SkipNode<E,K> (lev+1);
  y->data = e;
  for (int i = 0; i <= lev; i++) {
    // 插入到第 i 级链
    y->link[i] = last[i]->link[i];
    last[i]->link[i] = y;
  }

  return *this;
}

```

程序7-10所给出的代码可用来删除一个值为 k 的元素，并把所删除的元素放入 e 中。若没有值为 k 的元素，则引发 BadInput 异常。while 循环用来修改 Levels 的值，以找到一个至少包含一个元素的级（除非跳表为空）。若跳表为空，则 Levels 置为 0。

程序7-10 从跳表中删除元素

```

template<class E, class K>
SkipList<E,K>& SkipList<E,K>::Delete(const K& k, E& e)
{ // 删除与 k 相匹配的元素，并将所删除的元素放入 e
  // 如果不存在与 k 匹配的元素，则引发异常 BadInput
  if (k >= TailKey) throw BadInput(); // 关键值太大

  // 检查是否存在与 k 相匹配的元素
  SkipNode<E,K> *p = SaveSearch(k);
  if (p->data != k) throw BadInput(); // 不存在
}

```

```

// 从跳表中删除节点
for (int i = 0; i <= Levels && last[i]->link[i] == p; i++)
    last[i]->link[i] = p->link[i];

// 修改级数
while (Levels > 0 && head->link[Levels] == tail)
    Levels--;

e = p->data;
delete p;
return *this;
}

```

### 7.3.6 复杂性

当跳表中有  $n$  个元素时，搜索、插入、删除操作的复杂性均为  $O(n + \text{MaxLevel})$ 。在最坏情况下，可能只有一个  $\text{MaxLevel}$  级元素，且余下的所有元素均在 0 级链上。 $i > 0$  时，在  $i$  级链上花费的时间为  $\Theta(\text{MaxLevel})$ ，而在 0 级链上花费的时间为  $O(n)$ 。尽管最坏情况下的性能较差，但跳表仍不失为一种有价值的数据描述方法。其每种操作（搜索、插入、删除）的平均复杂性均为  $O(\log n)$ ，其证明超出了本书的范围。

至于空间复杂性，注意到最坏情况下所有元素都可能是  $\text{MaxLevel}$  级，每个元素都需要  $\text{MaxLevel} + 1$  个指针。因此，除了存储  $n$  个元素（也就是  $n * \text{sizeof}(\text{element})$ ），还需要存储链指针（所需空间为  $O(n * \text{MaxLevel})$ ）。不过，一般情况下，只有  $n * p$  个元素在 1 级链上， $n * p^2$  个元素在 2 级链上， $n * p^i$  在  $i$  级链上。因此指针域的平均值（不包括头尾节点的指针）是  $n_i p^i = n / (1 - p)$ 。因此虽然最坏情况下空间需求比较大，但平均的空间需求并不大。当  $p = 0.5$  时，平均空间需求（加上  $n$  个节点中的指针）大约是  $2n$  个指针的空间。

### 练习

4. 既然跳表中 0 级链是已排好序的，因此跳表可以支持顺序访问，返回每一个元素的时间为  $\Theta(1)$ 。在 `SkipList` 类中增加顺序访问函数 `Begin` 和 `Next`，分别返回字典中第一个元素的指针和下一个元素的指针（元素按从小到大次序排列），在没有第一个或下一个元素时，二者均应返回 0。每个函数的复杂度均应为  $\Theta(1)$ 。试测试代码的正确性。

5. 编写一个级的分配程序，采用本节中所介绍的把随机数的取值范围划分为若干段的策略。

6. 修改类 `SkipList` 以允许有相同值的元素出现。每个链从左到右按递增次序排列。用合适的数据测试代码。

7. 扩充类 `SkipList`，增加删除最小值、最大值元素的函数，以及按升序输出元素的函数。每个函数的复杂性分别是多少？

## 7.4 散列表描述

### 7.4.1 理想散列

字典的另一种描述方法就是散列（hash），它是用一个散列函数（hash function）把关键字

映射到散列表 (hash table) 中的特定位置。在理想情况下, 如果元素  $e$  的关键字为  $k$ , 散列函数为  $f$ , 那么  $e$  在散列表中的位置为  $f(k)$ 。要搜索关键字为  $k$  的元素, 首先要计算出  $f(k)$ , 然后看表中  $f(k)$  处是否有元素。如果有, 便找到了该元素。如果没有, 说明该字典中不包含该元素。在前一种情况中, 如果要删除该元素, 只需把表中  $f(k)$  位置置为空即可。在后一种情况中, 可以通过把元素放在  $f(k)$  位置以实现插入。

例7-5 考察例7-1中的学生记录字典。假设不用学生名, 而是用学生ID号(为六位整数)作为关键字。在一个班中, 假设最多有100个学生, 他们的ID号在951000和952000之间。函数  $f(k)=k-951000$  把学生ID号映射到0到1000之间。采用元素类型为E的数组  $ht[1001]$  作为散列表, 该表被初始化为0, 即对于  $0 \leq i \leq 1000$ , 有  $ht[i].key=0$ 。要搜索关键字为  $k$  的元素, 需计算  $f(k)=k-951000$ 。如果元素的关键字域不为0, 则此元素就在  $ht[f(k)]$  中。如果为0, 则字典中没有该元素。在后一种情况下, 可以把该元素插入到相应位置。前一种情况下可以通过把  $ht[f(k)].key$  置为0从而实现删除。

在理想情况下, 初始化一个空字典需要的时间为  $\Theta(b)$  ( $b$  为散列表中位置的个数), 搜索、插入、删除操作的时间均为  $\Theta(1)$ 。尽管理想的散列方法在许多场合都适用, 但还有许多应用因为关键字变化范围太大而不能创建一个这样的散列表。例如例7-1中, 每个学生的名字被截为最多12个字母长, 大写字母由相应的小写字母来代替, 特殊的字如连字符要被删掉。若现在用截短了的名字作为关键字。不够12个字母长的关键字可以在其前面加空格以补足12个。每个关键字可以被映射到相应的数值型关键字,  $a$  对应1,  $b$  对应2,  $\dots$ ,  $z$  对应26。这时关键字的范围为  $1$  到  $27^{12}-1$  (zzzzzzzzzzzz), 范围太大, 因此不能像理想散列方法那样建立数组  $ht$ 。

## 7.4.2 线性开型寻址散列

### 1. 方法

当关键字的范围太大, 不能用理想方法表示时, 可以采用比关键字范围小的散列表以及把多个关键字映射到同一位置的散列函数。虽然有多种函数映射方法, 但最常用的还是除法映射。除法散列函数的形式如下:

$$f(k) = k \% D \quad (7-2)$$

其中  $k$  为关键字,  $D$  是散列表的大小(即位置数),  $\%$  为求模操作符。散列表中的位置号从0到  $D-1$ , 每一个位置称为桶(bucket)。若关键字不是正整数类型(如 `int`, `long`, `char`, `unsigned char` 等), 则在计算  $f(k)$  之前必须把它转换成非负整数。对于一个长字符串, 可以采用取其2个字母或4个字母而变成无符号整数或无符号长整数的方法。 $f(k)$  是存储关键字为  $k$  的元素的起始桶(home bucket)。在良性情况下, 起始桶中所存储的元素即是关键字为  $K$  的元素。

图7-2a 中给出一个散列表  $ht$ , 桶号从0到10。表中有3个元素, 除数  $D$  为11。因为  $80 \% 11 = 3$ , 则80的位置为3,  $40 \% 11 = 7$ ,  $65 \% 11 = 10$ 。每个元素都在相应的桶中。散列表中余下的桶为空。

现在假设要在表中插入58。58的起始桶为  $f(58) = 58 \% 11 = 3$ 。这个桶已被另一个数占用, 这时就发生了碰撞(collision)。一般说来, 一个桶中可以存储多个元素, 因此发生碰撞也没什么了不起的。存储桶中若没有空间时就发生溢出(overflow)。但在我们的表中, 每个桶只能存储一个元素, 因此碰撞和溢出会同时发生。那么把58插到哪儿呢? 最简单的方法就是把58存储到表中下一个可用的桶中, 这种解决溢出的方法叫作线性开型寻址(linear open addressing)。

可把58存储在4号桶中。假设下一个要插入的元素值为24,  $24 \% 11$  为2。2号桶为空, 则把

24放入2号桶中。此时散列表如图 7-2b 所示。现在要插入 35。35的起始桶已满。采用线性开型寻址，结果如图 7-2c 所示。最后一例，插入 98。其起始桶 10 已满，则插入下一个可用桶 0 中。由此看来，在寻找下一个可用桶时，表被视为环形的。

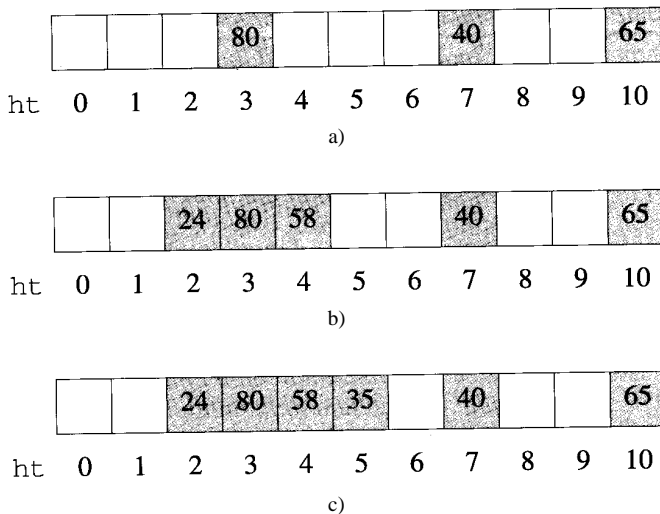


图7-2 散列表

刚才讲述了怎样用线性开型寻址法进行插入，现在介绍如何对这样一个散列表进行搜索。首先搜索起始桶  $f(k)$ ，接着对表中后继桶进行搜索，直到发生以下情况：1) 存有关键字为  $K$  的桶已找到，即找到了要搜索的元素；2) 到达一个空桶；3) 又回到  $f(k)$  桶。若发生后两种情况，则说明表中没有关键字为  $k$  的元素。

在完成一次删除操作后，必须能保证上述的搜索过程仍能够正常进行。若在图 7-2c 中删掉 58，不能仅仅把 4 号桶置为空，否则就无法找到关键字为 35 的元素。删除会带来多个元素的移动。可从欲删除的元素开始逐个检查每个桶以确定要移动的元素，直到到达一个空桶或返回删除操作所对应的桶为止。实现删除的另一种策略是为每个桶增加一个 `NeverUsed` 域。在表初始化时，该域被置为 `true`。当桶中存入一个元素时，`NeverUsed` 域被置为 `false`。这时搜索的结束条件 2) 变成：桶的 `NeverUsed` 域为 `true`。实现删除时，只需把表中相应位置置为空即可。一个新元素可被插入到从其对应的起始桶开始所找到的第一个空桶中。注意在这种方案中，`NeverUsed` 不会被重置为 `true`。用不了多长时间，所有的 (或几乎所有的) 桶的 `NeverUsed` 域均会被置为 `false`，这时搜索过程可能需要检查所有的桶。在这种情况下，为了提高性能，必须重新组织该表。比如，可把所有余下的元素都插入到一个空的散列表中。

## 2. C++实现

程序 7-11 给出了采用线性开型寻址的散列表的类定义。在类定义中假定散列表中每个元素的类型为 `E`，每个元素都有一个类型为 `K` 的 `key` 域。`key` 是用来计算起始桶的，因此类型 `K` 必须能够适应取模操作 `%`。散列表使用了两个数组，`ht` 和 `empty`。当且仅当 `ht[i]` 中不含有元素时，`empty[i]` 为 `true`。程序 7-12 给出了构造函数的代码。

程序 7-11 散列表的 C++ 类定义

```
template<class E, class K>
```

```
class HashTable {
public:
    HashTable(int divisor = 11);
    ~HashTable() {delete [ ] ht; delete [ ] empty;}
    bool Search(const K& k, E& e) const;
    HashTable<E,K>& Insert(const E& e);
private:
    int hSearch(const K& k) const;
    int D; // 散列函数的除数
    E *ht; // 散列数组
    bool *empty; // 一维数组
};
```

程序7-12 HashTable的构造函数

```
template<class E, class K>
HashTable<E,K>::HashTable(int divisor)
{ // 构造函数
    D = divisor;

    // 分配散列数组
    ht = new E [D];
    empty = new bool [D];

    // 将所有桶置空
    for (int i = 0; i < D; i++)
        empty[i] = true;
}
```

程序7-13给出了搜索函数。共享成员函数 Search 在没有找到关键字值为 k 的元素时返回 false，否则返回 true。并且若找到该元素，则在参数 e 中返回该元素。Search 函数调用了私有成员函数 hSearch。在满足如下三种情形之一时，hSearch 用来返回 b 号桶：1) empty[b] 为 false 且 ht[b] 的关键字值为 k；2) 表中没有关键字值为 k 的元素，empty[b] 为 true，可把关键字值为 k 的元素插入到 b 号桶中；3) 表中没有关键字值为 k 的元素，empty[b] 为 false，ht[b] 的关键字值不等于 k，且表已满。

程序7-13 查询函数

```
template<class E, class K>
int HashTable<E,K>::hSearch(const K& k) const
{ // 查找一个开地址表
    // 如果存在，则返回k的位置
    // 否则返回插入点（如果有足够空间）
    int i = k % D; // 起始桶
    int j = i; // 在起始桶处开始
    do {
        if (empty[j] || ht[j] == k) return j;
        j = (j + 1) % D; // 下一个桶
    }
```



```

    } while (j != i); // 又返回起始桶?

    return j; // 表已经满
}

template<class E, class K>
bool HashTable<E,K>::Search(const K& k, E& e) const
{// 搜索与k匹配的元素并放入e
// 如果不存在这样的元素, 则返回 false
    int b = hSearch(k);
    if (empty[b] || ht[b] != k) return false;
    e = ht[b];
    return true;
}

```

程序7-14给出了Insert函数的实现代码。函数从一开始就调用hSearch。若hSearch返回的*i*号桶为空, 则表中没有关键字为*k*的元素, 可以把该元素*e*插入到该桶中。若返回的桶非空, 则要么在桶中已包含了关键字为*k*的元素, 要么表已满。在前一种情况下, Insert函数引发一个BadInput异常。在后一种情况引发一个NoMem异常。练习14要求编写Delete函数的代码。

程序7-14 散列表的插入

```

template<class E, class K>
HashTable<E,K>& HashTable<E,K>::Insert(const E& e)
{// 在散列表中插入
    K k = e; // 抽取key值
    int b = hSearch(k);

    // 检查是否能完成插入
    if (empty[b]) {empty[b] = false;
        ht[b] = e;
        return *this;}

    // 不能插入, 检查是否有重复值或表满
    if (ht[b] == k) throw BadInput(); // 有重复
    throw NoMem(); // 表满
}

```

当E为用户自定义的类或数据类型时, 则有必要重载如%, !=, ==等操作符。程序7-7给出了重载的例子。

### 3. 性能分析

这里只分析时间复杂性。设*b*为散列表中桶的个数。散列函数中*D*为除数且*b=D*。初始化表的时间为 $\Theta(b)$ 。当表中有*n*个元素时, 最坏情况下插入和搜索时间均为 $\Theta(n)$ 。当所有*n*个关键字值都在同一个桶中时即出现最坏情况。通过比较散列在最坏情况下的复杂性与线性表在最坏情况下的复杂性, 可以看到二者完全相同。

但散列的平均性能还是相当好的。用 $U_n$ 和 $S_n$ 来分别表示在一次成功搜索和不成功搜索中平

均搜索的桶的个数。对于线性开型寻址，有如下公式成立：

$$U_n \sim \frac{1}{2} \left[ 1 + \frac{1}{(1-\alpha)^2} \right]$$

$$S_n \sim \frac{1}{2} \left[ 1 + \frac{1}{1-\alpha} \right]$$

其中  $\alpha = n/b$  为负载因子 (loading factor)。

所以若  $\alpha = 0.5$ ，则在不成功搜索时平均搜索的桶的个数为 2.5 个，成功搜索时则为 1.5 个。当  $\alpha = 0.8$  时，为 50.5 和 5.5。此时假设  $n$  至少为 51。当负载因子为 0.5 时，采用线性开型寻址散列表的平均性能要比线性表好得多。

#### 4. 确定 $D$

在实际应用过程中， $D$  的选择对于散列的性能有着重大的影响。当  $D$  为素数或  $D$  没有小于 20 的素数因子时，可以使性能达到最佳 ( $D$  等于桶的个数  $b$ )。

为了确定  $D$  的值，首先要了解影响成功搜索和不成功搜索性能的因素。通过  $U_n$  和  $S_n$  的公式，可以确定最大的  $\alpha$  值。根据  $n$  的值 (或是估计值) 和  $\alpha$  的值，可以得到  $b$  的最小许可值。然后找到一个比  $b$  大的最小的整数，该整数要么是素数，要么没有小于 20 的素数因子。这个整数即可作为  $D$  和  $b$  的值。

例 7-6 设计一个有近 1000 个元素的散列表。要求成功搜索时平均搜索的桶的个数不得超过 4，不成功搜索时平均搜索的桶的个数不超过 50.5。由  $U_n$  的公式，可得到  $\alpha = 0.9$ ，由  $S_n$  的公式，可以得到  $4 = 0.5 + 1/(2(1-\alpha))$  或  $\alpha = 6/7$ 。因此， $\alpha = \min\{0.9, 6/7\} = 6/7$ 。因此  $b$  最小为  $\lceil (7n/6) \rceil = 1167$ 。 $b = D = 37 \times 37 = 1369$ ，应为一个比较好的值 (虽然不是最小的选择)。

另一种计算  $D$  的方法是首先根据散列表的最大空间来确定  $b$  的最大可能值，然后取  $D$  为不大于这个最大值的整数，该整数要么是素数，要么没有小于 20 的素数因子。例如，如果在表中最多可以分配 530 个桶，则  $D$  和  $b$  的最佳选择为 23 (因  $23 \times 23 = 529$ )。

### 7.4.3 链表散列

#### 1. 方法

当散列发生溢出时，链表是一种好的解决方法。图 7-3 给出了散列表在发生溢出时采用链表来进行解决的方法。在上一个例子中，散列函数的除数为 11。在该散列表的组织中，每个桶仅含有一个节点指针，所有的元素都存储在该指针所指向的链表中。

在搜索关键字值为  $k$  的元素时，首先要计算其起始桶。起始桶号为  $k \% D$ ，然后搜索该桶所对应的链表。在插入时，首先要保证表中不含有相同关键字的元素。当然，此时的搜索仅限于该元素的起始桶所对应的链表。由于每次插入都要首先进行一次搜索，因此把链表按照升序排列比无序排列会更有效。最后，为了删除关键字值为  $k$  的元素，首先访问起始桶对应的链表，找到该元素，然后删除。

#### 2. C++ 实现

可以用一组有序链表来实现链表散列，见程序 7-15。由于该类引用了 SortedChain 类的成员，所以必须在类型  $E$  上定义操作符 `cout`, `==`, `!=` 和 `<`。在程序 7-7 中给出了操作符重载的范例。

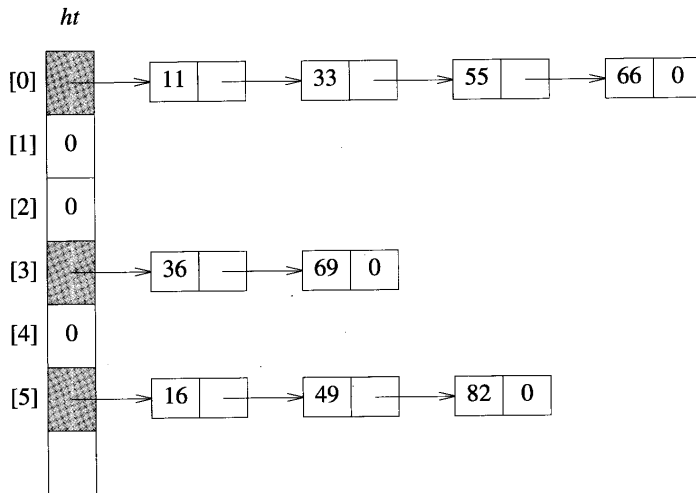


图7-3 链表散列

程序7-15 链表散列的类定义

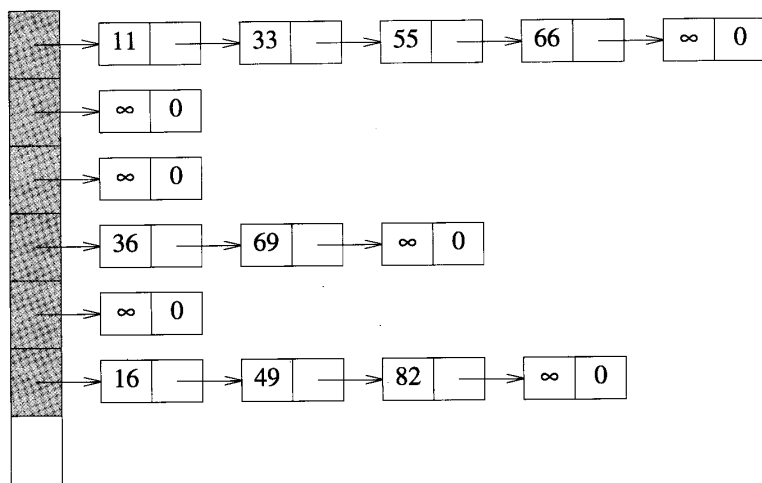
```

template<class E, class K>
class ChainHashTable {
public:
    ChainHashTable(int divisor = 11)
    {D = divisor;
     ht = new SortedChain<E,K> [D];}
    ~ChainHashTable() {delete [] ht;}
    bool Search(const K& k, E& e) const
    {return ht[k % D].Search(k, e);}
    ChainHashTable<E,K>& Insert(const E& e)
    {ht[e % D].DistinctInsert(e);
     return *this;}
    ChainHashTable<E,K>& Delete(const K& k, E& e)
    {ht[k % D].Delete(k, e);
     return *this;}
    void Output() const; // 输出散列表
private:
    int D; // 位置数
    SortedChain<E,K> *ht; // 链表数组
};

```

### 3. 一种改进方法

在图7-3所示的每条链表中加上一个尾节点，可以稍稍改进程序的性能。尾节点的关键字值最起码要比散列中所有元素的关键字值都大。在图7-4中尾节点的关键字用 来表示。在实际应用中，当关键字为整数时，可以采用在 limits.h 文件中定义的 INT\_MAX 常量。由于有了尾节点，在 SortedChain 的定义中出现的  $i \ \&\& \ i \rightarrow \text{data}$  均可改为  $i \rightarrow \text{data}$ 。在图7-4中每条链有不同的尾节点。在实际实现过程中，所有的链表可共用同一个尾节点。



表示非常大的关键字

图7-4 带尾节点的链表散列

#### 4. 与线性开型寻址比较

把线性开型寻址与没有尾节点的链表散列进行比较。令  $s$  为每个元素需占用的空间 (以字节为单位), 每个指针和每个整数类型的变量各占 2 个字节空间。同时, 设散列表中有  $b$  个桶和  $n$  个元素。首先注意到当使用线性开型寻址时有  $n \leq b$ , 而使用链表时  $n$  可能大于  $b$ 。

采用线性开型寻址所需要的空间为  $b(s+2)$  个字节, 其中  $s$  为每个元素占用的字节数。而使用链表所需要的空间为  $2b+2n+ns$  字节。当  $n < bs/(s+2)$  时, 链表方法所占用的空间要比开型寻址少。注意, 如果在实现线性开型寻址散列时采用一种节约空间的方法, 二者间的比较结果可能会发生变化。比如把数组 `empty` 压缩为  $b/8$  个字节 (在程序实现中该数组占用了  $2b$  个字节)。另外, 如果已知关键字的范围并非整个整数范围时, 可以使用一个不可能用到的整数 ( $-1$  或 `INT_MAX`) 作为空桶的关键字。

在最坏情况下, 用两种方法进行搜索时, 都要求搜索所有的  $n$  个元素。链表散列的平均搜索次数可用如下方法计算。对一条有  $i$  个节点的有序链表的一次不成功搜索可能要搜索 1, 2, 3, ... 第  $i$  个节点, 其中  $i \geq 0$ 。设每种情况发生的概率都相同, 则一次不成功搜索所要搜索的节点数为:

$$\frac{1}{i} \sum_{j=1}^i j = \frac{i(i+1)}{2i} = \frac{i+1}{2}$$

其中  $i \geq 1$ 。当  $i=0$  时平均搜索节点数为 0。对于链表散列, 假定链表的平均长度为  $n/b=\alpha$ 。当  $\alpha \geq 1$  时可用  $\alpha$  代替上面公式中的  $i$ , 从而可以得到:

$$U_n \sim \frac{\alpha+1}{2}, \alpha \geq 1$$

当  $\alpha < 1$  时, 由于链表的平均长度为  $\alpha$ , 搜索次数不可能比节点数多, 因此  $U_n \sim \alpha$ 。

计算  $S_n$  时, 需要知道  $n$  个标识符距其链表头节点的平均距离。为了计算距离, 假定各标识符是按升序插入的。当插入第  $i$  个标识符时, 其所在链表的长度为  $(i-1)/b$ 。由于标识符按升序插入, 所以第  $i$  个标识符被插入到相应链表的尾部。因此对该标识符的搜索需查找  $1+(i-1)/b$  个节点。还需要注意到, 由于标识符是按升序插入的, 它与链表头节点的距离并不随以后的插入

操作而改变。假定每个标识符被搜索的概率都相同，则有

$$S_n = \frac{1}{n} \sum_{i=1}^n \{1 + (i-1)/b\} = 1 + \frac{n-1}{2b} \sim 1 + \frac{\alpha}{2}$$

把采用链表的公式与采用线性开型寻址的公式相比较，可以看到使用链表时的平均性能要优于使用线性开型寻址。例如，当  $\alpha=0.9$  时，在链表散列中的一次不成功搜索，平均要检查 0.9 个元素，而一次成功搜索需要检查 1.45 个元素。对于线性开型寻址来说，不成功搜索时需要检查 50.5 个元素，成功时也需要检查 5.5 个元素。

### 5. 与跳表比较

跳表和散列均使用了随机过程来提高字典操作的性能。在使用跳表时，插入操作用随机过程来决定一个元素的级数。这种级数分配不考虑要插入元素的值。在散列中，当对不同元素进行插入时，散列函数随机地为不同元素分配桶，但散列函数需要使用元素的值。

通过使用随机过程，跳表和散列操作的平均复杂性分别为对数时间和常数时间。跳表的最坏时间复杂性为  $\Theta(n + \text{MaxLevel})$ ，而散列的最坏时间复杂性为  $\Theta(n)$ 。跳表中指针平均占用的空间约为  $\text{Maxlevel} + n/(1-p)$ ，在最坏情况下可能相当大。链表散列的指针所占用的空间为  $D+n$ 。

不过，跳表比散列更灵活。例如，只需简单地沿着 0 级链就可以在线性时间内按升序输出所有的元素。而采用链表散列时，需要  $\Theta(D+n)$  时间去收集  $n$  个元素并且需要  $O(n \log n)$  时间进行排序，之后才能输出。对于其他的操作，如查找或删除最大或最小元素，散列可能要花费更多的时间（仅考虑平均复杂性）。

## 练习

8. 用理想散列来实现字典的 C++ 类。假定元素的关键字是介于  $0 \sim \text{MaxKey}$  之间的整数， $\text{MaxKey}$  是在创建字典时由用户确定的。用适当的数据测试代码的正确性。

9. 在理想散列的某些应用中，尽管有足够的空间来构建关键字范围很大的散列，但由于初始化散列所需要的时间  $\Theta(b)$  太大以至于这种方法不切实际。例如，若关键字的范围为 1 000 000，而我们只需执行 100 个操作，百万单位的时间将花费在初始化上，而花在操作本身上的时间仅为 100 个单位。

针对这种应用，可以对理想散列的方法进行修改，采用两个数组：ht 和 ele。ht 是从 0 到  $\text{MaxKey}$  的整数数组 ( $\text{MaxKey}$  是最大可能的关键字)；ele 数组的数据类型为 E，其元素个数由要插入数组的不同关键字数决定（一般来说比  $\text{MaxKey}$  要小得多）。两个数组都不需要初始化。

由于不会被删除，ele 中的位置可被依次编号为 0, 1, 2, ...。计数器 LastE 表示上次所用到的 ele 中的位置，其初值为 -1。ht[j] 中的值可能无效也可能是 ele 的索引（用来寻找关键字为 j 的元素）。

把上述思想应用于理想散列的初始化、搜索、插入和删除函数。每个函数的复杂性应为  $\Theta(1)$ 。这些函数应作为类 IdealHashTable 的共享成员函数。试测试代码的正确性。

10. 指出在线性开型寻址散列中进行顺序访问所存在的困难。

11. 分别用公式化线性表和线性开型寻址编写字典程序。在该练习中不必编写删除函数。假设关键字为整数且  $D$  为 961。在表中插入随机产生的  $n=500$  个不同的整数，并在每  $m$  次插入后进行搜索。测出每次搜索的平均时间。从该练习中能得出什么结论？

12. 采用线性开型寻址，在下列各种情况下找到散列函数除数  $D$  的合适的值。

1)  $n=50$ ,  $S_n=3$ ,  $U_n=20$ 。

2)  $n=500$ ,  $S_n=5$ ,  $U_n=60$ 。

3)  $n=10$ ,  $S_n=2$ ,  $U_n=10$ 。

13. 对于以下各种条件, 找出散列函数除数  $D$  的合适的值。假设采用线性开型寻址。

1) MaxElements 530。

2) MaxElements 130。

3) MaxElements 150。

\*14. 编写类 HashTable(见程序 7-11) 的共享成员函数 Delete 的代码。不要改变类中其他成员函数。代码的最坏时间复杂性是多少? 用合适的数据测试代码的正确性。

15. 编写采用线性开型寻址的散列表的类定义, 要求在执行删除操作时采纳 NeverUsed 思想。为每个函数编写完整的 C++ 代码。当 60% 空桶的 NeverUsed 域为 false 时, 要重新组织散列表。编写重新组织散列表的代码, 重新组织过程在必要时需要移动元素, 并且对每个空桶, 其 NeverUsed 域必须置为 true。试测试代码的正确性。

16. 指出在链表散列中进行顺序访问所存在的困难。

17. 开发新的 SortedChainWithTail 类, 为每个有序链表增加一个尾节点。在操作开始时, 把欲搜索、插入或删除的元素或关键字放入尾节点中以简化代码。比较有尾节点和没有尾节点时的运行性能。

18. 从低层开发 ChainHashTable 类。定义自己的类 HashNode, 其中包含 data 域和 link 域, 不要使用链表类的任何版本。测试代码。

19. 从类 SortedChainWithTail(见练习 17) 中派生出类 ChainHashTable。比较该版本与练习 18 中 ChainHashTable 版本的性能。

20. 开发类 ChainHashWithTail, 每个散列链表皆为有尾节点的有序链表。所有链表的尾节点在物理上为同一个。比较该类与类 ChainHashTable(见程序 7-15) 的运行性能。

21. 为了简化链表散列的插入和删除操作, 可以在每个链表中加一个头节点。头节点是对上文中尾节点的补充。所有的插入和删除都在头、尾节点之间进行, 因此不会在链表的头部进行插入和删除操作。

1) 所有链表是否能使用同一个头节点? 为什么?

2) 在头节点的关键字域设置一个特定的值是否合理? 为什么?

3) 编写一个新的链表散列类定义, 要求使用头节点和尾节点。编写所有函数的代码。

4) 用合适的数据测试代码的正确性。

5) 指出带头、尾节点, 只带尾节点和没有头、尾节点这三种情况的优点和缺点。你会推荐哪种? 为什么?

## 7.5 应用——文本压缩

为了节约空间, 常常需要把文本文件采用压缩编码方式存储。例如, 一个包含 1000 个  $x$  的字符串和 2000 个  $y$  的字符串的文本文件在不压缩时占用的空间为 3002 字节 (每个  $x$  或  $y$  占用一个字节, 2 个字节用来表示串的结尾)。同样是这个文本文件, 采用游程长度编码 (run-length coding), 可存储为字符串 1000 $x$ 2000 $y$ , 仅为 10 个字母, 占用 12 个字节。若采用二进制表示游程长度 (1000 和 2000) 可以进一步节约空间。如果每个游程长度占用 2 个字节, 则可表示的最大游程长度为  $2^{16}$ , 这样上例中的字符串只需用 8 个字节来存储。当要读取编码文件时, 需对其进行解码。由压缩器 (compressor) 对文件进行编码, 由解压器 (decompressor) 进行解码。



在本节中，采用由Lempel、Ziv 和Welch 所开发的技术，来设计对文本文件进行压缩和解压缩的C++ 代码。这种技术被称为LZW方法，该方法相对简单，采用了理想散列和链表散列。

### 7.5.1 LZW压缩

LZW压缩方法把文本的字符串映射为数字编码。首先，为该文本文件中所有可能出现的字母分别分配一个代码。例如，要压缩的文本文件为：

aaabbbbbbaabaaba

字符串由a 和b 组成。为a 分配代码0，为b 分配代码1。字符串和编码的映射关系存储在字典中。每个字典的入口有 2 个域：key 和 code。由code 表示的字符串存在域key中。本例的初始字典由图7-5的前两列给出（也就是代码0和1）。

code	0	1	2	3	4	5	6	7
key	a	b	aa	aab	bb	bbb	bbba	aaba

图7-5 aaabbbbbbaabaaba的LZW压缩字典

若初始字典如图7-5所示，LZW压缩器不断地在输入文件中寻找在字典中出现的最长的前缀p，并输出其相应的代码。若输入文件中下一个字符为c，则为pc分配下一个代码，并插入字典。这种策略称为LZW规则。

用LZW方法来压缩上例字符串。文件中第一个在字典中出现的最长前缀为a，输出其编码0。然后为字符串aa 分配代码2，并插入到字典中。余下字符串中在字典内出现的最长前缀为aa，输出aa 对应的代码2，同时为字符串aab 分配代码3，并插入到字典中。注意，虽然为aab 分配的代码为3，但仅输出aa 的代码2。后缀b 将作为下一个代码的组成部分。不输出3是因为编码表不是压缩文件的组成部分。相反，在解压时，编码表由压缩文件重新构造。只要采用 LZW 规则，这种重建就是可能的。

紧接2之后，输出b 对应的代码。为bb 分配4，并插入字典中。然后输出bb 的编码，为bbb 分配代码5并插入字典中。输出5，并为bbba 分配编码6，然后插入字典中。接下来输出aab 的代码3，同时为aaba 分配代码7并插入字典。因此上例中字符串的编码为0214537。

### 7.5.2 LZW压缩的实现

#### 1. 输入和输出

压缩器的输入为文本文件，而输出为二进制文件。为了简单起见，假设输入文件名中不包含“.”（即文件名中没有扩展名）。如果输入文件名为InputFile，则输出文件名为InputFile.zzz。同时假设用户在命令行中输入要压缩的文件名。若压缩程序为Compress，则命令行为

Compress text

就可以得到文件text的压缩版本text.zzz。若用户没有输入文件名就应提醒用户输入。

函数SetFiles为输入和输出创建输入输出流（见程序7-16）。它假定函数main的原型为：

```
void main (int argc, char * argv[ ])
```

并假定in和out为全局变量，类型分别为 ifstream和ostream。argc是命令行中参数的个数，argv[i]为指向第i个参数的指针。若命令行为

Compress text

则argc为2，argv[0]指向字符串Compress，argv[1]指向text。

程序7-16 建立输入输出流

```
void SetFiles(int argc, char* argv[])
```



```

// 创建输入流和输出流
char OutputFile[50], InputFile[50];
// 检查是否提供了文件名
if (argc >= 2) strcpy(InputFile, argv[1]);
else { // 没有提供文件名, 提示用户输入
    cout << "Enter name of file to compress"
        << endl;
    cout << "File name should have no extension" << endl;
    cin >> InputFile;}

// 文件名不应有扩展名
if (strchr(InputFile, '.')) {
    cerr << "File name has extension" << endl;
    exit(1);}

// 以二进制方式打开文件
in.open(InputFile, ios::binary);
// in.open(InputFile); //对于g++而言
if (in.fail()) {cerr << "Cannot open " << InputFile << InputFile << endl;
    exit(1);}
strcpy(OutputFile, InputFile);
strcat(OutputFile, ".zzz");
out.open(OutputFile, ios::binary);
}

```

## 2. 组织字典

字典中每个元素有2个域: code和key。code为整型, 而key为长字符串。然而, 每个长度  $l > 1$  的关键字的前  $l-1$  个字符(称为关键字前缀)都可在字典中找到。因为每个字典入口都有一个不同的代码(不仅只是有唯一的关键字), 因此可以用代码代替其关键字前缀。在图7-5的例子中, 关键字aa可以替换为0a, 而aaba可以替换为3a。现在字典的形式如图7-6所示。

code	0	1	2	3	4	5	6	7
key	a	b	0a	2b	1b	4b	5a	3a

图7-6 aabbbbbbaabaaba修改后的LZW压缩字典

为简化对压缩文件的解码, 应对每个代码采用相同的位。更进一步, 可假设每个代码为12位长。因此可以最多分配  $2^{12}=4096$  个代码。由于每个字符为8位, 因此关键字可以用长整数(32位)来表示。低8位用来表示关键字中的最后一个字符, 后12位用来表示其前缀。字典本身可以表示为链表散列。若素数  $D=4099$  作为散列函数的除数, 存储密度就会小于1, 因为字典中最多有4096个入口。声明

```
ChainHashTable < element, unsigned long > h(D)
```

足以用来建立字典表。在我们的应用程序中, 没有使用与 ChainHashTable类的成员函数相关的Delete函数。

## 3. 输出代码

因为每个代码为12位, 每个字符为8位, 所以只能输代码的一部分作为一个字符。先输出代码的前8位, 余下4位留待以后输出。当要输出下一个代码时, 加上先前余下4位, 共16位可

作为2个字符输出。程序7-17给出输出函数的C++代码。mask1为255，mask2为15，excess为4，ByteSize为8。当还有余位待输出时status为1。此时，余下的4位放在变量Leftover中。

程序7-17 输出代码

```
void Output(unsigned long pcode)
{
    // 输出8 位, 余下的位保存在 LeftOver中
    unsigned char c,d;
    if (status) { // 余下4位
        d = pcode & mask1; //右边ByteSize位
        c = (LeftOver << excess) | (pcode >> ByteSize);
        out.put(c);
        out.put(d);
        status = 0;
    }
    else {
        LeftOver = pcode & mask2; // 右边多余的位
        c = pcode >> excess;
        out.put(c);
        status = 1;
    }
}
```

#### 4. 压缩

程序7-18给出LZW压缩算法的代码。首先用256个(alpha=256)8位字符及其代码对字典进行初始化，变量used用来保存目前已用的代码。因为每个代码为12位，则最多可分配4096个代码。在字典中找最长前缀，从长度为1, 2, 3...开始,直到发现某一个字符不在字典中为止。同时输出前缀对应的代码,并为该字符串分配一个新的代码(除非4096个代码全部用完)。

程序7-18 LZW压缩器

```
void Compress()
{
    // Lempel-Ziv-Welch压缩器
    // 定义并初始化代码字典
    ChainHashTable<element, unsigned long> h(D);
    element e;
    for (int i = 0; i < alpha; i++) { // 初始化
        e.key = i;
        e.code = i;
        h.Insert(e);
    }
    int used = alpha; // 所使用的代码数目

    // 输入并压缩
    unsigned char c;
    in.get(c); // 输入文件的第一个字符
    unsigned long pcode = c; // 前缀代码
    if (!in.eof()) { // 文件长度 > 1
        do { // 处理文件的其余部分
            in.get(c);
```

```

    if (in.eof()) break; // 完成
    unsigned long k = (pcode << ByteSize) + c;
    // 检查 k 的代码是否已经在字典中
    if (h.Search(k, e)) pcode = e.code; // 在字典中
    else { // k 不在表中
        Output(pcode);
        if (used < codes) // 创建新的代码
        {
            e.code = used++;
            e.key = (pcode << ByteSize) | c;
            h.Insert(e);
        }
        pcode = c;
    } while(true);

    // 输出最后一个 (部分) 代码
    Output(pcode);
    if (status) {c = LeftOver << excess; out.put(c);}
}

out.close();
in.close();
}

```

## 5. 头文件及main函数

程序7-19给出了压缩程序的头文件、常量定义、数据类型、全局变量和 main函数。

程序7-19 压缩程序的main函数

```

#include <fstream.h>
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include "chash.h"

// 常量
const D = 4099, // 散列函数的除数
      codes = 4096, // 2^12
      ByteSize = 8,
      excess = 4, // 12 - ByteSize
      alpha = 256, // 2^ByteSize
      mask1 = 255, // alpha - 1
      mask2 = 15; // 2^excess - 1

class element {
    friend void Compress();
public:
    operator unsigned long() const {return key;}
    element& operator =(unsigned long y)
    {key = y; return *this;}
}

```

```
private:
    int code;
    unsigned long key;
};

int LeftOver, // 尚未输出的代码位
    status = 0; // 0 意味着在 LeftOver 中没有未输出的位
ifstream in;
ofstream out;
void main ( int argc , char* argv [ ] )
{
    SetFiles (argc, argv);
    Compress ( ) ;
}
```

### 7.5.3 LZW解压缩

解压时要输入代码，然后用代码所表示的文本来替换这些代码。代码到文本的映射可按下面方法重新构造。首先把分配给单一字母的代码插入字典中。象前面一样，字典的入口为代码-文本对。然而此时是根据给定的代码，去寻找相应入口（而不是根据文本）。压缩文件中的第一个代码对应于一个单一的字母，因此可以由该字母代替。对于压缩文件中的其他代码  $p$ ，要考虑到两种情况：1) 在字典中；2) 不在字典中。当  $p$  在字典中时，找到与  $p$  相关的文本  $text(p)$  并输出。并且，由压缩器原理可知，若在压缩文件中代码  $q$  写在  $p$  之前且  $text(q)$  是与  $q$  对应的文本，则压缩器会为文本  $text(q)$ （其后紧跟着  $fc(p)$ ， $text(p)$  的第一个字符）分配一新代码。因此在字典中插入序偶（下一个代码， $text(q)fc(p)$ ）。情况2）只有在当前文本段形如  $text(q)text(q)fc(q)$  且  $text(p)=text(q)fc(q)$  时才会发生。相应的压缩文件段是  $qp$ 。在压缩的过程中，为  $text(q)fc(q)$  分配的代码为  $p$ 。在解压过程中，在用  $text(q)$  代替  $q$  后，又遇到代码  $p$ 。然而，此时字典中没有与  $p$  对应的文本。因为这种情况只在解压文本段为  $text(p)text(q)fc(q)$  时才会发生，因此可以对  $p$  解码。当遇到一个没有定义代码文本对的代码  $p$  时， $p$  对应的文本为  $text(q)fc(q)$ ，其中  $q$  为  $p$  前面的代码。

用此解码策略来解压前面的例子。字符串 aaabbbbbaabaaba 被压缩为代码 0214537。首先，初始化字典，在其中插入 (0,a) 和 (1,b) 后，得到图 7-5 字典的前两个入口。压缩文件的第一个代码为 0，则应用 a 代替。下一个代码 2 未定义。因为前一个代码为 0，且  $text(0)=a$ ， $fc(0)=a$ ，则  $text(2)=text(0)fc(0)=aa$ 。因此用 aa 代替 2，并把 (2, aa) 插入字典中。下一个代码 1 由 b 来替换且把 (3,  $text(2)fc(1)$ )=(3, aab) 插入字典中。下一代码 4 不在字典中。其前面的代码为 1，则  $text(4)=text(1)fc(1)=bb$ 。把 (4, bb) 插入字典中，且在解压文件中输出 bb。当遇到下一个代码 5 时，(5, bbb) 被插入字典中，同时把 bbb 输出到解压文件中。再下一代码为 3， $text(3)=aab$  则把 aab 输出，并将序偶 (6,  $text(5)fc(3)$ )=(6, bbba) 插入字典。当遇到 7 时，把 (7,  $text(3)fc(3)$ )=(7, aaba) 插入字典中并输出 aaba。

### 7.5.4 LZW解压缩的实现

#### 1. 输入/输出

函数 Setfiles(见程序 7-20) 与压缩器中的对应函数功能相同。它输入解压文件的名称，并加上 .zzz 的扩展名以得到相应的压缩文件名。

程序7-20 建立输入输出流

```

void SetFiles(int argc, char* argv[])
{
    // 确定文件名
    char OutputFile[50], InputFile[50];

    // 检查是否提供了文件名
    if (argc == 2) strcpy(OutputFile, argv[1]);
    else {
        // 没有提供文件名, 提示用户输入
        cout << "Enter name of file to decompress"
              << endl;
        cout << "Omit the extension .zzz" << endl;
        cin >> OutputFile;
    }

    // 文件名中不应带扩展名
    if (strchr(OutputFile, '.'))
        {cerr << "File name has extension" << endl;
         exit(1);}

    strcpy(InputFile, OutputFile);
    strcat(InputFile, ".zzz");

    // 以二进制方式打开文件
    in.open(InputFile, ios::binary);
    // in.open(InputFile) 对g++而言
    if (in.fail()) {cerr << "Cannot open "
                    << InputFile << endl;
                  exit(1);}
    out.open(OutputFile, ios::binary);
    // out.open(OutputFile) 对g++而言
}

```

## 2. 组织字典

因为可根据所给的代码查询字典并且代码总数为 4096, 所以可以采用数组  $ht[4096]$ , 把  $text(p)$  存储在  $ht[p]$  中。像理想散列那样使用数组  $ht$ , 散列函数  $f(k)=k$ 。同时像图 7-6 那样,  $text(p)$  可以存储为其前缀代码和最后一个字符 (后缀)。在解压过程中, 把  $text(p)$  的前缀和后缀分别作为整数和字符存储非常方便。因此若  $text(p)=text(q)c$ , 则  $ht[p].suffix$  字符为  $c$ ,  $ht[p].code$  等于  $q$ 。

当采用这种字典组织方式时, 见程序 7-21, 可从最后一个字符  $ht[p].suffix$  开始, 按从右到左的次序来构建  $text(q)$ 。程序从表  $ht$  中得到代码大于  $\alpha$  的后缀值。  $text(p)$  被收集到数组  $s[]$  中, 然后被输出。由于  $text(p)$  是从右到左存储的, 因此  $text(p)$  的第一个字符存储在  $s[size]$  中。

程序7-21 计算text(code)

```

void Output(int code)
{
    // 输出与代码相对应的串
    size = -1;
    while (code >= alpha) {
        // 字典中的后缀
        s[++size] = ht[code].suffix;
    }
}

```

```
code = ht[code].prefix;
}
s[++size] = code; // code < alpha

// 解压所得的串为 s[size] ... s[0]
for (int i = size; i >= 0; i--)
    out.put(s[i]);
}
```

### 3. 输入代码

由于12位代码在压缩文件中是按8位字节顺序表示的, 所以要把函数 Output (见程序7-17)的输出结果转换过来。可由 GetCode函数(见程序7-22)来完成这种转换。此处唯一的新常量是 mask, 其值为15, 它可以帮助我们得到一个字节的低4位。

程序7-22 在压缩文件中提取代码

```
bool GetCode (int& code)
{// 把压缩文件中的下一个代码取入 code
// 如果没有代码, 则返回 false
    unsigned char c, d;
    in.get(c); // 输入8位
    if (in.eof()) return false; // 没有代码

    // 检查上一次是否有剩余的位
    // 如果有, 则取用其中的4位
    if (status) code = (LeftOver << ByteSize) | c;
    else { // 如果没有剩余的位, 则需要另外4位
        in.get(d); // 再取8位
        code = (c << excess) | (d >> excess);
        LeftOver = d & mask; // 保存余下的4位
    }
    status = 1 - status;
    return true;
}
```

### 4. 解压缩

程序7-23给出了LZW解压器。压缩文件的第一个代码在 while 循环体外解码, 而其他代码则在循环体内解码。因为压缩文件中的第一个代码总是在 0到alpha之间, 因此它仅表示一个字符, 并可以通过把整数转换成无符号字符得到。在每个 while循环的开始, s[size]中存有上次输出的解码文本的第一个字符。为了使第一个循环也满足此条件, 可以把 size置为0且s[0]置为压缩文件中第一个代码所对应的唯一的一个字符。

while 循环体不停地从压缩文件中得到代码 ccode并对其解码。ccode可能有以下两种情况: 1) 在字典中。2) 不在字典中。当且仅当 ccode < used时, ccode在字典中, 其中 ht[0:used]是 ht表的已定义部分。在这种情况下用函数 Output和LZW规则解码, 产生一新代码, 其后缀是刚输出的与 ccode相对应的文本的第一个字母。当 ccode没有定义时, 即本节开始所讨论的情况, ccode的代码是 text (pcode)s[size]。可根据此信息为 code创建字典的入口并输出与其相对应的解码后的文本。

程序7-23 LZW解压器

```
void Decompress()
{
    // 解压一个压缩文件
    int used = alpha; //迄今所使用的代码

    // 输入并解压缩
    int pcode, // 前一个代码
        ccode; // 当前代码
    if (GetCode(pcode)){ // 文件不为空
        s[0] = pcode; // pcode 代表的字符
        out.put(s[0]); // 输出pcode对应的串
        size = 0; // s[size] 是所输出的最后一个串的第一个字符

        while(GetCode(ccode)) { // 取另外的代码
            if (ccode < used) { // ccode已定义
                Output(ccode);
                if (used < codes) { // 创建新代码
                    ht[used].prefix = pcode;
                    ht[used++].suffix = s[size];}}
            else { // 特殊情况, 未定义代码
                ht[used].prefix = pcode;
                ht[used++].suffix = s[size];
                Output(ccode);}
            pcode = ccode;}
        }

    out.close();
    in.close();
}
```

## 5. 头文件和main函数

程序7-24给出LZW解压缩程序所包含的头文件、常量、类型定义、函数原型和 main函数。

程序7-24 解压缩程序的main函数

```
#include <fstream.h>
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

class element {
    friend void Decompress();
    friend void Output(int);
private:
    int prefix;
    unsigned char suffix;
};
```



```
// 常量
const codes = 4096, // 2^12
      ByteSize = 8,
      excess = 4, // 12 - ByteSize
      alpha = 256, // 2^ByteSize
      mask = 15; // 2^excess - 1

// 全局变量
unsigned char s[codes]; // 用于重构文本
int size, // 重构文本的大小
    LeftOver, // 上一个代码的剩余位
    status = 0; // 当且仅当没有剩余位时，其值为 0
element ht[codes]; // 字典
ifstream in;
ofstream out;

void main ( int argc, char* argv[ ] )
{
    SetFiles ( argc, argv );
    Decompress ( );
}
```

## 练习

22. 用LZW压缩器产生的压缩文件是否有可能比原文件长呢？如果可能，长多少？
23. 为由以下字母 {a, b, ..., z, 0, 1, ..., 9, ., , , ;, : } 和换行符组成的文件编写一个LZW压缩器和解压器。试测试程序的正确性。压缩文件是否可能比原文件长？
24. 重新修改LZW压缩和解压缩程序，使得每当压缩/解压缩1024x个字节后，重新初始化代码表。取文本文件长为100K到200K之间，x=10, 20, 30, 40 和50。测试修改后的程序。采用哪种x值的压缩效果最好？

## 7.6 参考及推荐读物

跳表是由 William Pugh 提出的。其平均复杂性的分析见论文 Skip lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33, 6, 1990, 668~676。

对于 Lempel-Ziv 压缩方法的描述是基于 T. Welch 的论文，A Technique for High-Performance Data Compression. *IEEE Computer*, 1994.6, 8-19。要想得到更好的压缩数据参见 D. Lelewer, D. Hirschberg. Data Compression. *ACM computing Surveys*, 19, 3, 1987, 261~296。

China-pub.com

下载

## 第8章 二叉树和其他树

在一片丛林中有各种各样的树、植物和动物。在数据结构的世界中也有许多“树”，不过本书不可能全部介绍。在本章中将学习两种基本的树：一般树（简单树）和二叉树。第9、10和11章中对其他树有更详细的介绍。

在本章的应用部分给出了树的两个应用。第一个应用是关于在一个树形分布的网络中设置信号调节器。第二个应用是3.8.3节中所介绍的在线等价类问题。在线等价类问题在本章中又被称为合并/搜索问题。利用树来解决等价类问题要比3.8.3节中的链表解决方案高效得多。

另外，本章中还覆盖了以下内容：

- 树和二叉树的术语，如高度、深度、层、根、叶子、子节点、父节点和兄弟节点。
- 二叉树的公式化描述和链表描述。
- 4种常用的二叉树遍历方法：前序遍历，中序遍历，后序遍历和按层遍历。

### 8.1 树

到目前为止，我们已经介绍了线性数据结构和表数据结构。这些数据结构一般不适合于描述具有层次结构的数据。在层次化的数据之间可能有祖先-后代、上级-下属、整体-部分以及其他类似的关系。

例8-1 [Joe的后代] 图8-1给出了Joe的后代，并按层次方式组织，其中Joe在最顶层。Joe的孩子（Ann，Mary和John）列在下一层，在父母和孩子间有一条边。在层次表示中，非常容易地找到Ann的兄弟姐妹，Joe的后代，Chris的祖先等。

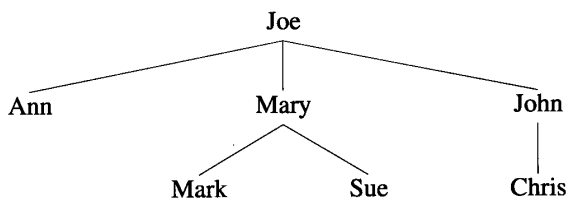


图8-1 Joe的后代

例8-2 [合作机构] 作为层次数据的一个例子，考虑图8-2的合作管理机构。在层次中地位最高的人（此处为总裁）在图中位置最高。在层次中地位次之的（即副总裁）在图中位于总裁之下等等。副总裁为总裁的下属，总裁是他们的上级。每个副总裁都有他自己的下属，而其下属又

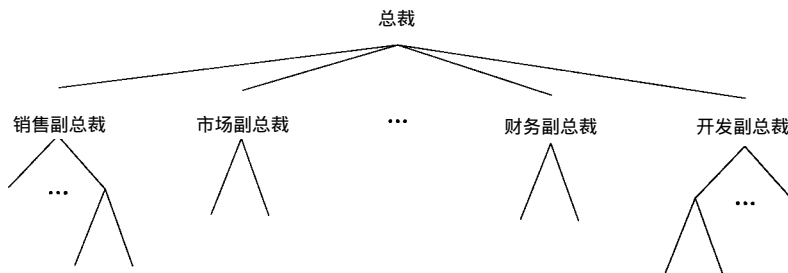


图8-2 合作管理结构

有他们自己的下属。在图中，在每个人与其直接下属或上级之间都有一条边互连。

例8-3 [政府机构] 图8-3是联邦政府各分支机构的层次图。在最顶层是整个联邦政府。层次结构的下一级，是其主要的隶属单位(例如不同的部)。每个部可进一步细分。这些分支在层次结构的下一级画出。例如，国防部分成陆军、海军、空军和海军陆战队。在每个机构及其分支机构间都有一条边。图8-3的数据即为整体-部分关系的例子。

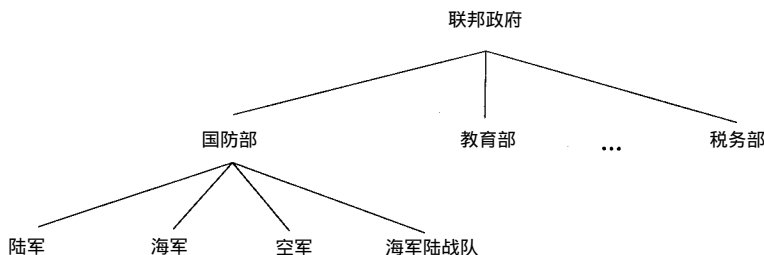


图8-3 联邦政府模型

例8-4 [软件工程] 考察另一种层次数据——软件工程中的模块化技术。通过模块化，可以把大的复杂的任务分成一组小的不太复杂的任务。模块化的目标是把软件系统分成许多功能不相关的部分或模块以便于进行相对独立的开发。由于解决几个小问题比解决大问题更容易一些，因此模块化方法可以缩短整个软件的开发时间。另外，不同的程序员可以同时开发不同的模块。如果有必要，每个模块可以再细分，从而得到如图 8-4所示的用树形表示的模块层次结构。该树给出了某文字处理器的一种可行的模块分解图。

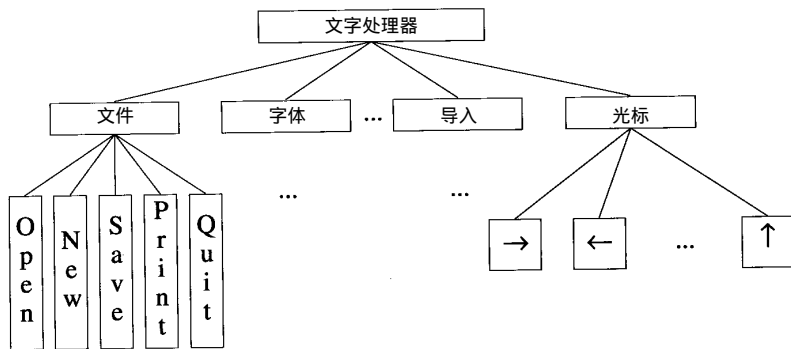


图8-4 文字处理器的模块层次结构

文字处理器的最顶层模块被划分为下一层的几个模块，在图 8-4中只给出4个。文件模块完成与文本文件有关的操作，如打开一个已存在文件（Open），打开一个新文件（New），保存文件（Save），打印文件（Print），从文字处理器中退出（Quit）。在层次结构中下一层的每一个模块分别代表一个函数。字体模块处理与字体有关的所有功能。这些功能包括改变字体、大小、颜色等。若把具有这些功能的模块在图中画出的话，那么它们一定出现在字体模块之下。导入模块用于处理图形、表格以及其他格式的文本文件。光标模块处理屏幕上光标的移动。在接口完全设计好后，程序员可以以相对独立的方式分析、设计和开发每个模块。

当一个软件系统以模块化方式划分好之后，可以很自然地以模块为单位来开发该系统。在最终完成的软件系统中，所含有的模块数与模块层次结构中节点数一样多。模块化可以促进对

欲解决问题的智能化管理。通过把一个大问题系统地分解成小而相对独立的小问题，可以使大问题的解决更省力。可以将独立的问题分配给不同的人同时解决。而在一个单一模块上进行分工是非常困难的。开发模块化软件的另一好处是，分开测试一些小而独立的模块比测试一个大的模块要容易得多。层次结构清晰地给出了模块间的关系。

定义 [树] 树 (tree)  $t$  是一个非空的有限元素的集合，其中一个元素为根 (root)，余下的元素 (如果有的话) 组成  $t$  的子树 (subtree)。

现在看一下定义与层次数据例子之间的关系。层次中最高层的元素为根。其下一级的元素是余下元素所构成的子树的根。

例8-5 在Joe的后代例子中(例8-1)，数据集合是{Joe, Ann, Mary, Mark, Sue, John, Chris}，因此 $n=7$ ，树的根为Joe。余下的元素被分成三个不相交的集合{Ann}，{Mary, Mark, Sue}和{John, Chris}。{Ann}是只有一个元素的树，其根为Ann。{Mary, Mark, Sue}的根为Mary，而{John, Chris}的根为John。集合{Mary, Mark, Sue}余下的元素分成不相交的集合{Mark}和{Sue}，二者均为单元素的子树，集合{John, Chris}余下的元素也为单元素子树。

在画一棵树时，每个元素都代表一个节点。树根在上面，其子树画在下面。在树根与其子树的根 (如果有子树) 之间有一条边。同样的，每一棵子树也是根在上，其子树在下。在一棵树中，边连结一个元素及其子节点。在图 8-1 中，Ann, Mary, John是Joe的孩子 (children)，Joe是他们的父母 (parent)。有相同父母的孩子为兄弟 (sibling)。Ann, Mary, John在图8-1的树中为兄弟，而Mark和Chris则不是。此外还有其他术语：孙子 (grandchild)，祖父 (grandparent)，祖先 (ancestor)，后代 (descendent) 等。树中没有孩子的元素称为叶子 (leaf)。因此在图8-1中，Ann, Mark, Sue 和Chris 是树的叶子。树根是树中唯一一个没有父节点的元素。

例8-6 在合作机构的例子中 (例8-2)，公司雇员是树中的元素。总裁是树的根，余下的分成不相交的集合，代表公司的不同分支，每个分支有一个副总裁，为该分支子树的根。分支中余下的元素分成不相交的集合，代表不同的部。部长是子树的根。余下元素同样可分成不同的科等。

副总裁是总裁的子节点，部长是副总裁的子节点。总裁是副总裁的父节点，每个副总裁是其分支中元素的父节点。

在图8-3中，根为联邦政府。其子树的根为国防部，教育部，...，税务部等。联邦政府是其子节点的父节点。国防部的子节点为陆军、海军、空军、海军陆战队。国防部的子节点之间为兄弟关系，同时它们也是叶节点。

树的另一常用术语为级 (level)。指定树根的级为1，其孩子 (如果有) 的级为2，孩子的孩子为3，等等。在图8-3中，联邦政府在第1级，国防部、教育部、税务部在第2级，陆军、海军、空军和海军陆战队在第3级。

元素的度 (degree of an element) 是指其孩子的个数。叶节点的度为0，在图8-4中文件模块的度为5。树的度 (degree of a tree) 是其元素度的最大值。

## 练习

1. 给出本书中主要元素的树形表示(整本书、章、节、小节)。

1) 树中共有多少个元素?

2) 标出叶节点。

3) 标出第3级元素。

4) 给出每个元素的度。

2. 访问 <http://www.cise.ufl.edu>, 即佛罗里达大学计算机、信息科学和工程系主页。通过连接到更下一级的网页, 绘出网页间的层次结构。用节点表示网页, 用线连接网页。

1) 此结构一定是一棵树吗? 为什么?

2) 若此结构是一棵树, 指出其根和叶子。

## 8.2 二叉树

定义 [二叉树] 二叉树 (binary tree)  $t$  是有限个元素的集合 (可以为空)。当二叉树非空时, 其中有一个称为根的元素, 余下的元素 (如果有的话) 被组成 2 个二叉树, 分别称为  $t$  的左子树和右子树。

二叉树和树的根本区别是:

- 二叉树可以为空, 但树不能为空。
- 二叉树中每个元素都恰好有两棵子树 (其中一个或两个可能为空)。而树中每个元素可有若干子树。

• 在二叉树中每个元素的子树都是有序的, 也就是说, 可以用左、右子树来区别。而树的子树间是无序的。

像树一样, 二叉树也是根节点在顶部。二叉树左 (右) 子树中的元素画在根的左 (右) 下方。在每个元素和其子节点间有一条边。

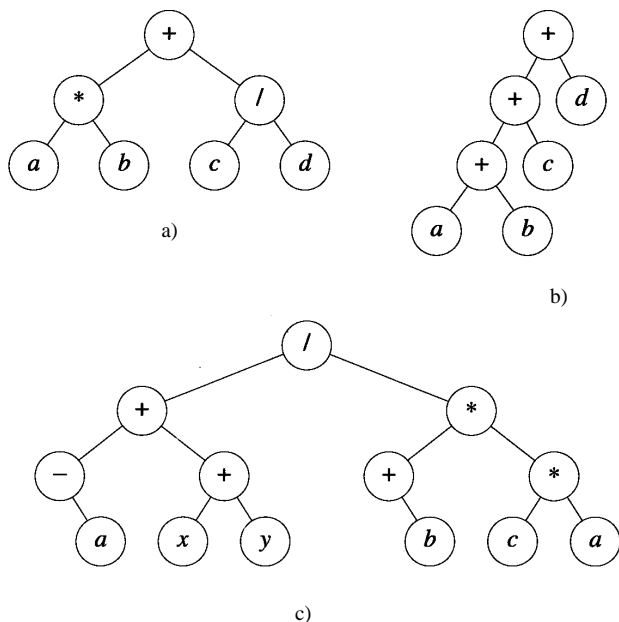


图8-5 数学表达式树

a)  $(a * b) + (c / d)$     b)  $((a + b) + c) + d$     c)  $((-a) + (x + y)) / ((+b) * (c * a))$

图8-5给出了表示数学表达式的二叉树。每个操作符（+，-，\*，/）可以有一个或两个操作数。左操作数是操作符的左子树，右操作数是操作符的右子树。树中的叶节点为常量或变量。注意在数学表达式树中没有括号。

数学表达式二叉树的一个应用是产生一个表达式的优化代码。我们并不研究怎样从数学表达式树中产生最优代码的算法，只是用它来说明许多操作可以用二叉树来解决。

### 练习

3. 1) 标出图8-5中二叉树的叶子。
- 2) 标出图8-5b 中第3级的所有节点。
- 3) 在图8-5c 中第4级有多少个节点？
4. 给出如下各表达式的二叉树：
  - 1)  $(a + b) / (c - d * e) + e + g * h / a$ 。
  - 2)  $-x - y * z + (a + b + c / d * e)$ 。
  - 3)  $((a + b) > (c - e)) || a < f \&\& (x < y || y > z)$ 。

## 8.3 二叉树的特性

特性1 包含 $n$  ( $n > 0$ )个元素的二叉树边数为 $n - 1$ 。

证明 二叉树中每个元素（除了根节点）有且只有一个父节点。在子节点与父节点间有且只有一条边，因此边数为 $n - 1$ 。

二叉树的高度（height）或深度（depth）是指该二叉树的层数。图8-5a 中二叉树的高度为3，而图8-5b 和c 的高度为4。

特性2 若二叉树的高度为 $h$ ， $h \geq 0$ ，则该二叉树最少有 $h$ 个元素，最多有 $2^h - 1$ 个元素。

证明 因为每一层最少要有1个元素，因此元素数最少为 $h$ 。每元素最多有2个子节点，则第 $i$ 层节点元素最多为 $2^i - 1$ 个， $i > 0$ 。 $h = 0$ 时，元素的总数为0，也就是 $2^0 - 1$ 。当 $h > 0$ 时，元素的总数不会超过 $\sum_{i=1}^h 2^{i-1} = 2^h - 1$ 。

特性3 包含 $n$ 个元素的二叉树的高度最大为 $n$ ，最小为 $\lceil \log_2(n+1) \rceil$ 。

证明 因为每层至少有一个元素，因此高度不会超过 $n$ 。由特性2，可以得知高度为 $h$ 的二叉树最多有 $2^h - 1$ 个元素。因为 $n \leq 2^h - 1$ ，因此 $h \geq \log_2(n+1)$ 。由于 $h$ 是整数，所以 $h \geq \lceil \log_2(n+1) \rceil$ 。

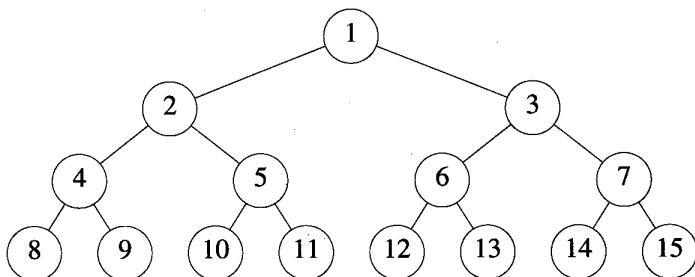


图8-6 高度为4的满二叉树



当高度为 $h$ 的二叉树恰好有 $2^h - 1$ 个元素时,称其为满二叉树(full binary tree)。图8-5a是一个高度为3的满二叉树。图8-5b和c的二叉树不是满二叉树。图8-6给出高度为4的满二叉树。

假设对高度为 $h$ 的满二叉树中的元素按从第上到下,从左到右的顺序从1到 $2^h - 1$ 进行编号(如图8-6所示)。假设从满二叉树中删除 $k$ 个元素,其编号为 $2^h - i, 1 \leq i \leq k$ ,所得到的二叉树被称为完全二叉树(complete binary tree)。图8-7给出三棵完全二叉树。注意满二叉树是完全二叉树的一个特例,并且,注意有 $n$ 个元素的完全二叉树的深度为 $\lceil \log_2(n+1) \rceil$ 。

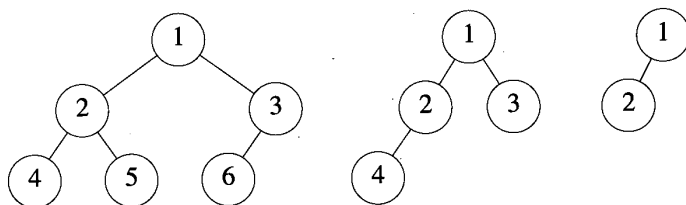


图8-7 完全二叉树

在完全二叉树中,一个元素与其孩子的编号有非常好的对应关系。其关系在特性4中给出。

特性4 设完全二叉树中一元素的序号为 $i, 1 \leq i \leq n$ 。则有以下关系成立:

- 1) 当 $i=1$ 时,该元素为二叉树的根。若 $i>1$ ,则该元素父节点的编号为 $i/2$ 。
- 2) 当 $2i > n$ 时,该元素无左孩子。否则,其左孩子的编号为 $2i$ 。
- 3) 若 $2i+1 > n$ ,该元素无右孩子。否则,其右孩子编号为 $2i+1$ 。

证明 通过对 $i$ 进行归纳即可得证。

## 练习

5. 证明特性4。

6. 在 $k$ 叉树中,每个节点最多有 $k$ 个孩子。其子节点分别称为该节点的第一个,第二个...,第 $k$ 个孩子。

- 1) 模拟特性1给出 $k$ 叉树的性质。
- 2) 模拟特性2给出 $k$ 叉树的性质。
- 3) 模拟特性3给出 $k$ 叉树的性质。
- 4) 模拟特性4给出 $k$ 叉树的性质。
7. 有 $m$ 个叶子的二叉树最多有多少个节点?

## 8.4 二叉树描述

### 8.4.1 公式化描述

二叉树的公式化描述利用了特性4。二叉树可以作为缺少了部分元素的完全二叉树。图8-8给出二叉树的两个样例。第一棵二叉树有三个元素(A、B和C),第二棵二叉树有五个元素(A、B、C、D和E)。没有涂阴影的圈表示缺少的元素。所有的元素(包括缺少的元素)按前面介绍的方法编号。

在公式化描述方法中,按照二叉树对元素的编号方法,将二叉树的元素存储在数组中。图

8-8同时给出了二叉树的公式化描述。缺少的元素由白圈和方格描述。当缺少很多元素时，这种描述方法非常浪费空间。实际上，一个有 $n$ 个元素的二叉树可能最多需要 $2^n - 1$ 个空间来存储。当每个节点都是其他节点的右孩子时，存储空间达到最大。图8-9给出这种情况下一棵有四个元素的二叉树，这种类型的二叉树称为右斜(right-skewed)二叉树。当缺少的元素数目比较少时，这种描述方法很有效。

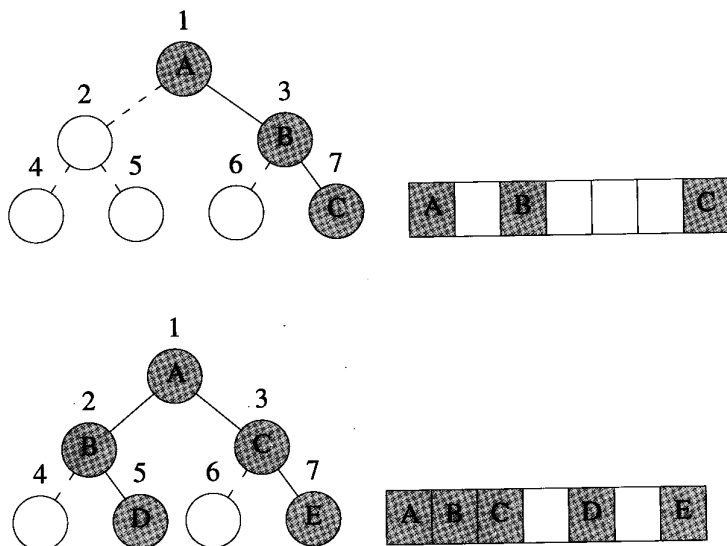


图8-8 不完全二叉树

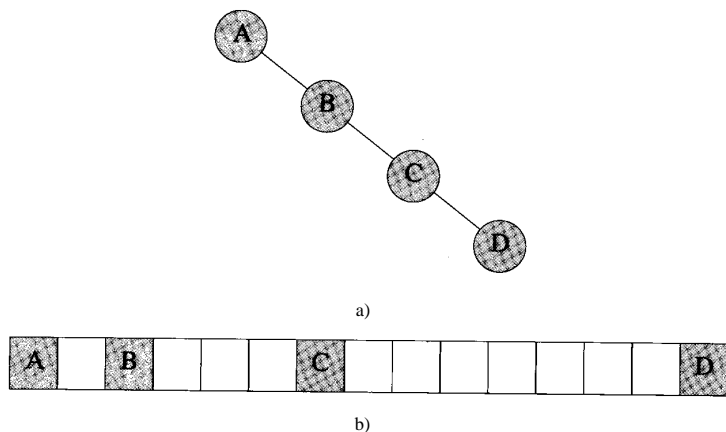


图8-9 右斜二叉树

a) 右斜树 b) 数组表示

#### 8.4.2 链表描述

二叉树最常用的描述方法是用链表或指针。每个元素都用一个有两个指针域的节点表示，这两个域为LeftChild和RightChild。除此两个指针域外，每个节点还有一个data域。可以像程序8-1那样把节点结构定义为C++类模板。这种定义为二叉树节点提供了三种构造函数。第一种

无参数，初始化时节点的左右孩子域被置为 0（即 NULL）；第二种有一个参数，可用此参数来初始化数据域，孩子域被置为 0；第三种有 3 个参数，可用来初始化节点的 3 个域。

二叉树的边可用一个从父节点到子节点的指针来描述。指针放在父节点的指针域中。因为包括  $n$  个元素的二叉树恰有  $n-1$  条边，因此将有  $2n-(n-1)=n+1$  个指针域没有值，这些域被置为 0。图 8-10 给出图 8-8 中二叉树的链表描述。

程序 8-1 链表二叉树的节点类

```
template <class T>
class BinaryTreeNode {
    friend void Visit(BinaryTreeNode<T> *);
    friend void InOrder(BinaryTreeNode<T> *);
    friend void PreOrder(BinaryTreeNode<T> *);
    friend void PostOrder(BinaryTreeNode<T> *);
    friend void LevelOrder(BinaryTreeNode<T> *);
    friend void main(void);
public:
    BinaryTreeNode() {LeftChild = RightChild = 0;}
    BinaryTreeNode(const T& e)
        {data = e;
         LeftChild = RightChild = 0;}
    BinaryTreeNode(const T& e, BinaryTreeNode *l,
                  BinaryTreeNode *r)
        {data = e;
         LeftChild = l;
         RightChild = r;}
private:
    T data;
    BinaryTreeNode<T> *LeftChild, //左子树
                      *RightChild; //右子树
};
```

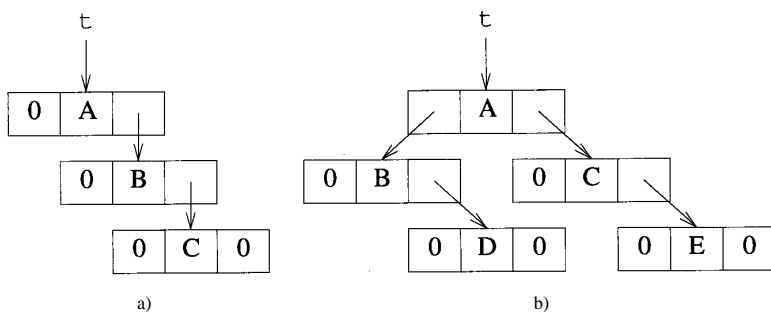


图 8-10 链表描述

用一个变量(在图 8-10 中为  $t$ ) 来保存二叉树的根，用该变量的名称来指称根节点或整个二叉树，因此可以说根节点  $t$  或二叉树  $t$ 。从根节点开始，沿着  $LeftChild$  和  $RightChild$  指针域逐层进行搜索，可以访问二叉树  $t$  中的所有节点。在二叉树中不设置指向父节点的指针一般不会有什问题，因为在二叉树的大部分函数中并不需要此指针。若某些应用需要此指针，可在每个节

点增加一个指针域。

## 8.5 二叉树常用操作

二叉树的常用操作为：

- 确定其高度。
- 确定其元素数目。
- 复制。
- 在屏幕或纸上显示二叉树。
- 确定两棵二叉树是否一样。
- 删除整棵树。
- 若为数学表达式树，计算该数学表达式。
- 若为数学表达式树，给出对应的带括号的表达式。

以上所有操作可以通过对二叉树进行遍历来完成。在二叉树的遍历中，每个元素仅被访问一次。在访问时执行对该元素的所有操作。这些操作包括：把该元素显示在屏幕或纸上；计算以该元素为根的子树所表示的数学表达式的值；对二叉树中元素的个数加 1；删除表示该元素的节点等。

## 8.6 二叉树遍历

有四种遍历二叉树的方法：

- 前序遍历。
- 中序遍历。
- 后序遍历。
- 逐层遍历。

前三种遍历方法在程序 8-2，8-3，8-4 中给出。假设要遍历的二叉树采用前一节所介绍的链表的方法来描述，并且 BinaryTreeNode 被定义为类或模板结构。

程序 8-2 前序遍历

---

```
template <class T>
void PreOrder(BinaryTreeNode<T> *t)
{// 对*t进行前序遍历
    if (t) {
        Visit(t);           // 访问根节点
        PreOrder(t->LeftChild); // 前序遍历左子树
        PreOrder(t->RightChild); // 前序遍历右子树
    }
}
```

---

程序 8-3 中序遍历

---

```
template <class T>
void InOrder(BinaryTreeNode<T> *t)
{// 对*t进行中序遍历
    if (t) {
```

---

```

InOrder(t->LeftChild);    // 中序遍历左子树
Visit(t);                 // 访问根节点
InOrder(t->RightChild);   // 中序遍历右子树
}
}

```

程序8-4 后序遍历

```

template <class T>
void PostOrder(BinaryTreeNode<T> *t)
{// 对*t进行后序遍历
    if (t) {
        PostOrder(t->LeftChild);    // 后序遍历左子树
        PostOrder(t->RightChild);   // 后序遍历右子树
        Visit(t);                   // 访问根节点
    }
}

```

在前三种方法中，每个节点的左子树在其右子树之前遍历。这三种遍历的区别在于对同一个节点在不同时刻进行访问。在进行前序遍历时，每个节点是在其左右子树被访问之前进行访问的；在中序遍历时，首先访问左子树，然后访问子树的根节点，最后访问右子树。在后序遍历时，当左右子树均访问完之后才访问子树的根节点。

图8-11给出程序8-2，8-3和8-4产生的结果。其中Visit(t)由cout << t->data代替。所输入的二叉树为图8-5所示的二叉树。

前序	$++ab/cd$	$+++abcd$	$/+-a+xy*+b*ca$
中序	$a*b+c/d$	$a+b+c+d$	$-a+x+y/+b*c*a$
后序	$ab*cd/+$	$ab+c+d+$	$a-xy++b+ca**/$
	a)	b)	c)

图8-11 二叉树按前序，中序，后序遍历的结果

当对一棵数学表达式树进行中序，前序和后序遍历时，就分别得到表达式的中缀、前缀和后缀形式。中缀（infix）形式即平时所书写的数学表达式形式，在这种形式中，每个二元操作符（也就是有两个操作数的操作符）出现在左操作数之后，右操作数之前。在使用中缀形式时，可能会产生一些歧义。例如， $x+y \times z$  可以理解为  $(x+y) \times z$  或  $x+(y \times z)$ 。为了避免这种歧义，可对操作符赋予优先级并采用优先级规则来分析中缀表达式。在完全括号化的中缀表达式中，每个操作符和相应的操作数都用一对括号括起来。更甚者把操作符的每个操作数也都用一对括号括起来。如  $((x)+(y))$ ， $((x)+((y)*(z)))$  和  $((x)+(y))*((y)+(z))*((w))$ 。这种表达形式可通过修改程序8-5的中序遍历算法得到。

程序8-5 输出完全括号化的中缀表达式

```

template <class T>
void Infix(BinaryTreeNode<T> *t)
{// 输出表达式的中缀形式
    if (t) {cout << '(';
        Infix(t->LeftChild);    // 左操作数

```

```
cout << t->data;           // 操作符
Infix(t->RightChild);       // 右操作数
cout << ');}

}
```

在后缀 (postfix) 表达式中, 每个操作符跟在操作数之后, 操作数按从左到右的顺序出现。在前缀 (prefix) 表达式中, 操作符位于操作数之前。在前缀和后缀表达式中不会存在歧义。因此, 在前缀和后缀表达式中都不必采用括号或优先级。从左到右或从右到左扫描表达式并采用操作数栈, 可以很容易确定操作数和操作符的关系。若在扫描中遇到一个操作数, 把它压入堆栈, 若遇到一个操作符, 则将其与栈顶的操作数相匹配。把这些操作数推出栈, 由操作符执行相应的计算, 并将所得结果作为操作数压入堆栈。

在逐层遍历过程中, 按从顶层到底层的次序访问树中元素, 在同一层中, 从左到右进行访问。由于遍历中所使用的数据结构是一个队列而不是栈, 因此写一个按层遍历的递归程序很困难。程序 8-6 用来对二叉树进行逐层遍历, 它采用了队列数据结构 (见 6.3 节中定义的一类 LinkedQueue)。队列中的元素指向二叉树节点。当然, 也可以采用公式化队列。

程序 8-6 逐层遍历

```
template <class T>
void LevelOrder(BinaryTreeNode<T> *t)
{ // 对*t逐层遍历
    LinkedQueue<BinaryTreeNode<T>*> Q;
    while (t) {
        Visit(t); // 访问 t

        // 将t的右孩子放入队列
        if (t->LeftChild) Q.Add(t->LeftChild);
        if (t->RightChild) Q.Add(t->RightChild);

        // 访问下一个节点
        try {Q.Delete(t);}
        catch (OutOfBounds) {return;}
    }
}
```

程序 8-6 中, 仅当树非空时, 才进入 while 循环。首先访问根节点, 然后把其子节点加到队列中。当队列添加操作失败时, 由 Add 引发 NoMem 异常, 由于没有捕获该异常, 因此当异常发生时 LevelOrder 函数将退出。在把 t 的子节点加入队列后, 要从队列中删除 t 元素。若队列为空, 则由 Delete 引发 OutOfBounds 异常, 这由 catch 语句来处理。因为一个空队列意味着遍历的结束, 所以执行 return 语句。若队列非空, 则 Delete 把所删除的元素返回至变量 t。被删除的元素指向下一个欲访问的节点。

设二叉树中元素数目为  $n$ 。这四种遍历算法的空间复杂性均为  $O(n)$ , 时间复杂性为  $\Theta(n)$ 。当 t 的高度为  $n$  时 (也就是图 8-9 给出的右斜二叉树的情况), 通过观察其前序、中序和后序遍历时所使用的递归栈空间即可得到上述结论。当 t 为满二叉树时, 逐层遍历所需要的队列空间为  $\Theta(n)$ 。每个遍历算法花在树中每个节点上的时间为  $\Theta(1)$  (假设访问一个节点的时间为  $\Theta(1)$ )。

## 练习

8. 编写公式化描述的二叉树的前序遍历程序。假设二叉树的元素存储在数组  $a$  中, 其中  $Last$  用于保存树中最后一个元素的位置。当位置  $i$  中没有元素时,  $a[i]=0$ 。给出该程序的时间复杂性。

9. 按中序遍历方法完成练习1。

10. 按后序遍历方法完成练习1。

11. 按逐层遍历方法完成练习1。

12. 编写一个C++函数, 用于复制一个公式化描述的二叉树。

13. 编写两个C++函数, 复制用 `BinaryTreeNode` 模板结构描述的二叉树  $t$ 。第一个函数按前序遍历整棵树, 第二个按后序遍历。两个函数所需要到的递归栈空间有什么不同?

14. 编写一个函数, 计算用模板结构 `BinaryTreeNode` 描述的表达式树的值。假设每个节点有一个 `value` 域, 常量和变量的 `value` 域内有其相应的值。

15. 编写一个函数删除二叉树  $t$  (提示: 采用后序遍历方法)。假设  $t$  是一链表树, 调用 C++ 函数 `delete` 释放节点空间。

16. 写一交互式进程按中序方法来遍历链表二叉树, 在程序中可采用公式化堆栈。尽量使进程做得比较完美。遍历需多少栈空间? 给出栈空间与节点数  $n$  间的关系。

17. 按前序遍历方法完成练习16。

18. 按后序遍历方法完成练习16。

\*19. 设  $t$  是数据域类型为 `int` 的二叉树, 每个节点的数据都不相同。数据域的前序和中序排列是否可唯一地确定这棵二叉树? 如果能, 给出一函数来构造此二叉树。指出函数的时间复杂性。

20. 按前序和后序遍历方法完成练习19。

\*21. 按中序和后序遍历方法完成练习19。

22. 编写一个C++函数, 输入后缀表达式, 构造其二叉树表示。设每个操作符有一个或两个操作数。

\*23. 用前缀表达式完成练习22。

24. 编写一个C++函数, 把后缀表达式转换为完全括号化的中缀表达式。

\*25. 用前缀表达式完成练习24。

\*26. 把一个表达式的中缀形式 (不一定是完全括号化的) 转换成后缀形式。在该练习中假定允许的操作符为二元操作符  $+$ 、 $-$ 、 $\times$ 、 $/$ , 允许的分界符为  $($  和  $)$ 。因为操作数的顺序在中缀、前缀、后缀中都是一样的, 所以在从中缀向后缀、前缀转换时, 仅从左到右扫描中缀表达式, 看到操作数时, 则直接输出, 而把操作符保留在栈中, 直到合适的时机输出 (具体的输出时机由操作符和分界符 (的优先级来确定)。假定  $+$  和  $-$  的优先级为1,  $\times$  和  $/$  的优先级为2。栈外的  $($  的优先级为3, 栈内的  $($  的优先级为0。

\*27. 完成练习26, 要求产生前缀表达式。

\*28. 完成练习26, 要求输出二叉树形式。

29. 编写一个函数, 计算以后缀表达式的值。假设表达式以数组方式描述。

## 8.7 抽象数据类型 `BinaryTree`

现在我们已明白什么是二叉树, 可以用抽象数据类型来描述二叉树 (见ADT8-1)。注意:



抽象数据类型的描述应独立于具体的实现形式。尽管我们希望在二叉树上进行的操作非常多，但这里只列出了几个常用的操作。在8-9节将对ADT 8-1进行扩充。

#### ADT8-1 二叉树的抽象数据类型描述

抽象数据类型 *BinaryTree*{

实例

元素集合；如果不空，则被划分为根节点、左子树和右子树；

每个子树仍是一个二叉树

操作

*Create* ()：创建一个空的二叉树；

*IsEmpty*：如果二叉树为空，则返回 true，否则返回 false

*Root* (x)：取x为根节点；如果操作失败，则返回 false，否则返回 true

*MakeTree* (root, left, right)：创建一个二叉树，root作为根节点，left作为左子树，right作为右子树

*BreakTree* (root, left, right)：拆分二叉树

*PreOrder*：前序遍历

*InOrder*：中序遍历

*PostOrder*：后序遍历

*LevelOrder*：逐层遍历

}

## 8.8 类BinaryTree

程序8-7给出了二叉树抽象数据类型的C++定义。此定义中采用了链接描述的二叉树，对应的C++类为BinaryTree。函数Visit作为遍历函数的参数，以利于不同操作的实现。

#### 程序8-7 二叉树类定义

```
template<class T>
class BinaryTree {
public:
    BinaryTree() {root = 0;};
    ~BinaryTree() {};
    bool IsEmpty() const
    {return ((root) ? false : true);}
    bool Root(T& x) const;
    void MakeTree(const T& element, BinaryTree<T>& left, BinaryTree<T>& right);
    void BreakTree(T& element, BinaryTree<T>& left, BinaryTree<T>& right);
    void PreOrder(void(*Visit) (BinaryTreeNode<T> *u))
    {PreOrder(Visit, root);}
    void InOrder(void (*Visit) (BinaryTreeNode<T> *u))
    {InOrder(Visit, root);}
    void PostOrder(void(*Visit) (BinaryTreeNode<T> *u));
    {Postorder (Visit, root); }
    void LevelOrder(void(*Visit) (BinaryTreeNode<T> *u));
private:
    BinaryTreeNode<T> *root; // 根节点指针
```

```

void PreOrder(void (*Visit) (BinaryTreeNode<T> *u) , BinaryTreeNode<T> *t);
void InOrder(void(*Visit) (BinaryTreeNode<T> *u) , BinaryTreeNode<T> *t);
void PostOrder(void(*Visit) (BinaryTreeNode<T> *u) , BinaryTreeNode<T> *t);
};

```

程序8-8给出了共享成员函数Root, MakeTree, BreakTree的代码, 而程序8-9和8-10给出私有遍历函数的代码。函数 MakeTree和BreakTree要求参与操作的三棵树应该互不相同。若它们相同的话, 则程序有可能得出错误结果。例如调用 Y.MakeTree(e,X,X)得到一棵二叉树, 其左右子树共享同样的节点。这种共享只有在 X为空二叉树时才正确。X.MakeTree(e,X,Y)将在返回之前把X.root(left.root)置为0。因此不管X, Y的初始值是什么, 在调用 MakeTree之后, X均为空二叉树。在练习30中, 要求给出能弥补以上缺陷的MakeTree和BreakTree版本。

程序8-8 共享成员函数的实现

```

template<class T>
bool BinaryTree<T>::Root(T& x) const
{// 取根节点的数据域, 放入 x
// 如果没有根节点, 则返回 false
if (root) {x = root->data;
            return true;}
else return false; // 没有根节点
}

template<class T>
void BinaryTree<T>::MakeTree(const T& element, BinaryTree<T>& left, BinaryTree<T>& right)
{// 将left, right和element 合并成一棵新树
// left, right和this必须是不同的树
// 创建新树
root = new BinaryTreeNode<T>
        (element, left.root, right.root);

// 阻止访问left和right
left.root = right.root = 0;
}

template<class T>
void BinaryTree<T>::BreakTree(T& element, BinaryTree<T>& left, BinaryTree<T>& right)
{// left, right和this必须是不同的树
// 检查树是否为空
if (!root) throw BadInput(); // 空树

// 分解树
element = root->data;
left.root = root->LeftChild;
right.root = root->RightChild;

delete root;
}

```

```
root = 0;
}
```

---

程序8-9 前序，中序和后序遍历

---

```
template<class T>
void BinaryTree<T>::PreOrder(void(*Visit)(BinaryTreeNode<T> *u), BinaryTreeNode<T> *t)
{// 前序遍历
    if (t) {Visit(t);
        PreOrder(Visit, t->LeftChild);
        PreOrder(Visit, t->RightChild);
    }
}

template <class T>
void BinaryTree<int>::InOrder(void(*Visit)(BinaryTreeNode<T> *u), BinaryTreeNode<T> *t)
{// 中序遍历
    if (t) {InOrder(Visit, t->LeftChild);
        Visit(t);
        InOrder(Visit, t->RightChild);
    }
}

template <class T>
void BinaryTree<T>::PostOrder(void(*Visit)(BinaryTreeNode<T> *u), BinaryTreeNode<T> *t)
{// 后序遍历
    if (t) {PostOrder(Visit, t->LeftChild);
        PostOrder(Visit, t->RightChild);
        Visit(t);
    }
}
```

---

程序8-10 逐层遍历

---

```
template<class T>
void BinaryTree<T>::LevelOrder(void(*Visit)(BinaryTreeNode<T> *u))
{// 逐层遍历
    LinkQueue<BinaryTreeNode<T>*> Q;
    BinaryTreeNode<T> *t;
    t = root;
    while (t) {
        Visit(t);
        if (t->LeftChild) Q.Add(t->LeftChild);
        if (t->RightChild) Q.Add(t->RightChild);
        try {Q.Delete(t);}
        catch (OutOfBounds) {return;}
    }
}
```

```

    }
}

```

程序8-11中使用了BinaryTree类。程序中构造了一个四节点的二叉树，并进行了前序遍历以确定树中的节点数目。

程序8-11 类BinaryTree的应用

```

#include <iostream.h>
#include "binary.h"

int count = 0;
BinaryTree<int> a,x,y,z;

template<class T>
void ct(BinaryTreeNode<T> *t) {count++;}

void main(void)
{
    y.MakeTree(1,a,a);
    z.MakeTree(2,a,a);
    x.MakeTree(3,y,z);
    y.MakeTree(4,x,a);
    y.PreOrder(ct);
    cout << count << endl;
}

```

## 8.9 抽象数据类型及类的扩充

本节将ADT8-1中的抽象数据类型进行扩充，增加如下二叉树操作：

- *PreOutput()*：按前序方式输出数据域。
- *InOutput()*：按中序方式输出数据域。
- *PostOutput()*：按后序方式输出数据域。
- *LevelOutput()*：逐层输出数据域。
- *Delete()*：删除一棵二叉树，释放其节点。
- *Height()*：返回树的高度。
- *Size()*：返回树中节点数。

### 8.9.1 输出

四个输出函数(前序输出、中序输出、后序输出、逐层输出)可以通过定义一个私有静态成员函数Output来实现，此静态成员函数的形式如下：

```

static void Output(BinaryTreeNode<T> *t)
{ cout << t->data << ' ';}

```

四个共享输出函数的形式如下：

```

void PreOutput( )

```

```

    { PreOrder(Output , root); cout << endl;}
void InOutput( )
    { InOrder(Output , root); cout << endl;}
void PostOutput( )
    {PostOrder(Output , root); cout << endl;}
void LevelOutput( )
    { LevelOrder(Output); cout << endl;}

```

因为Visit操作的时间复杂性为  $\Theta(1)$ ，对包括  $n$  个节点的二叉树来说，每种遍历方法所花费时间均为  $\Theta(n)$  (假设遍历成功)，因此每种输出方法的时间复杂性为  $\Theta(n)$ 。

### 8.9.2 删除

要删除一棵二叉树，需要删除其所有节点。可以通过后序遍历在访问一个节点时，把它删除。也就是说先删除左子树，然后右子树，最后删除根。共享成员函数 Delete 的形式如下：

```
void Delete( ) { PostOrder (Free , root); root = 0;}
```

其中Free为私有成员函数：

```
static void Free(BinaryTreeNode<T> *t) {delete t;}
```

当要删除的二叉树中有  $n$  个节点时，Delete 函数的时间复杂性为  $\Theta(n)$ 。

### 8.9.3 计算高度

通过进行后序遍历，可以得到二叉树的高度。首先得到左子树的高度  $hl$ ，然后得到右子树的高度  $hr$ 。此时，树的高度为：

```
max{hl , hr} + 1
```

不幸的是，不能用程序 8-9 中给出的后序遍历代码，因为在进行遍历时要有返回值 (也就是子树的高度)。为了实现共享成员函数 Height，在程序 8-7 的 public 部分增加如下语句：

```
int Height( ) const {return Height (root);}
```

在 private 部分增加：

```
int Height (BinaryTreeNode<T> *t) const;
```

私有成员函数 Height 在程序 8-12 中给出。其时间复杂性为  $\Theta(n)$ ，其中  $n$  是二叉树中的节点数。

程序 8-12 计算二叉树的高度

```

template <class T>
int BinaryTree<T>::Height(BinaryTreeNode<T> *t) const
{// 返回树*t的高度
    if (!t) return 0;                // 空树
    int hl = Height(t->LeftChild);    // 左子树的高度
    int hr = Height(t->RightChild);    // 右子树的高度
    if (hl > hr) return ++hl;
    else return ++hr;
}

```

### 8.9.4 统计节点数

可以用上述四种遍历方法中的任何一种来获取二叉树中的节点数。因为在每种遍历方法中对每个节点都仅访问一次，只要在访问每个节点时将一个全局计数器增加 1 即可。在程序 8-11 中，使用用户自定义函数 `ct` 来获得二叉树的节点数。通过把如下代码加入到程序 8-7 中的 `public` 部分，可以定义一个等价类成员函数 `Size`。

```
int Size ()
{
    _count = 0;
    PreOrder(Add1, root);
    return _count;
}
```

`_count` 是在类定义之外定义的一个整型变量，其定义如下：

```
int _count;
```

私有成员函数 `Add1` 的原型为：

```
static void Add1(BinaryTreeNode<T> *t) { _count++;}
```

`Size` 的时间复杂性为  $O(n)$ ，其中  $n$  为二叉树中节点数。

### 练习

30. 编写二叉树成员函数 `MakeTree` 和 `BreakTree` 的新版本，考察每个操作所用到的三棵树是否相同。如果有相同者，确定应采取什么措施并给出函数代码。

31. 1) 扩充抽象数据类型 *BinaryTree*，增加 `Compare(X)` 操作，用来比较该二叉树与二叉树  $X$ 。若两棵二叉树相同返回 `true`，否则返回 `false`。

2) 扩充 C++ 类 *BinaryTree*，增加用于二叉树比较的共享成员函数，并测试代码。

32. 1) 扩充抽象数据类型 *BinaryTree*，增加复制二叉树的操作 `Copy()`。若此操作失败，引发一个合适的异常。

2) 扩充 C++ 类 *BinaryTree*，增加共享成员函数 `Copy`。测试代码的正确性。

\*33. 从类 *BinaryTree* 派生子类 *Expression*，该类中包括以下操作

- 1) 输出完全括号化的中缀表达式。
- 2) 输出前缀和后缀表达式。
- 3) 从前缀形式转换成表达式树。
- 4) 从后缀形式转换成表达式树。
- 5) 从中缀形式转换成表达式树。
- 6) 计算表达式树的值。

用适当数据测试代码的正确性。

## 8.10 应用

### 8.10.1 设置信号放大器

在一个分布网络中，资源被从生产地送往其他地方。例如，汽油或天然气经过管道网络从汽油/天然气生产基地输送到消耗地。同样的，电力也是通过电网从发电厂输送到各消耗点。可以用术语信号 (signal) 来指称所输送的资源(汽油、天然气、电力等)。当信号在网络中传输时，其性能的某一个或几个方面可能会有所损失或衰减。例如在传输过程中，天然气的气压会减少，电的电压会降低。另一方面，当信号在网络中传输时，噪声会增加。在信号从信号源到

消耗点传输过程中, 仅能容忍一定范围内的信号衰减。为了保证信号衰减不超过容忍值 (tolerance), 应在网络中合适的位置放置信号放大器 (signal booster)。信号放大器可以增加信号的压强或电压使其与源端的相同; 可以增强信号, 使信号与噪声之比与源端的相同。在本节中, 将设计一个算法以确定把信号放大器放在何处。目标是要使所用的放大器数目最少并且保证信号衰减(与源端信号相关)不超过给定的容忍值。

为简化问题, 设分布网络是一树形结构, 源是树的根。树中的每一节点 (除了根) 表示一个可以用来放置放大器的子节点, 其中某些节点同时表示消耗点。信号从一个节点流向其子节点。图8-12给出一树形分布网络。每条边上标出从父节点到子节点的信号衰减量。信号衰减的单位可认为是附加剂。在图8-12中信号从节点 $p$ 流到节点 $v$ 的衰减量为5。从节点 $q$ 到节点 $x$ 的衰减量为3。

设 $d(i)$ 表示节点 $i$ 与其父节点间的衰减量。因此, 在图8-12中,  $d(w)=2$ ,  $d(p)=0$ ,  $d(r)=3$ 。因为信号放大器只能放在树的节点上, 若节点 $i$ 的 $d(i)>$ 容忍值, 则不可能通过放置放大器来使信号的衰减不超过容忍值。例如, 若容忍值为1, 则在图8-12中是不可能 $p$ 和 $r$ 间通过放置放大器使衰减量小于等于1的。

对任一节点 $i$ , 设 $D(i)$ 为从节点 $i$ 到以 $i$ 为根节点的子树的任一叶子的衰减量的最大值。若 $i$ 为叶节点, 则 $D(i)=0$ 。图8-12中, 当 $i \in \{w, x, t, y, z\}$ 时,  $D(i)=0$ 。对于其他节点,  $D(i)$ 可以用下式来表示:

$$D(i) = \max_{j \text{ 是 } i \text{ 的一个孩子}} \{D(j) + d(j)\}$$

因此 $D(s)=2$ 。在此公式中, 要计算节点的 $D$ 值, 必须先计算其子节点的 $D$ 值。因而必须遍历整棵树, 先访问子节点然后访问父节点。这样当访问一个节点时, 就可同时计算其 $D$ 值。这种遍历方法是高度大于2的树的后序遍历的一种自然扩充。

假设按照上面方法, 在计算 $D$ 的过程中, 遇到一节点 $i$ , 且其有一子节点 $j$ 满足 $D(j)+d(j)>$ 容忍值。若不在 $j$ 处放置放大器, 则从 $i$ 节点到叶节点的信号衰减量将会超过容忍值, 即使在 $i$ 处放置放大器也是如此。例如在图8-12中, 当计算 $D(q)$ 时, 有 $D(s)+d(s)=4$ 。若容忍值为3, 则在 $q$ 点或其祖先的任意一点放置放大器, 并不能减少 $q$ 与其后代间的衰减量。必需在 $s$ 点放一个放大器或在其子节点放一个或多个放大器。若在节点 $s$ 处放一放大器, 则 $D(q)=2$ 。

计算 $D$ 并放置放大器 $s$ 的伪代码为:

```
D(i)=0;
for (i 的每个孩子 j)
    if ((D(j)+d(j))>tolerance) 在 j 放置放大器;
    else D(i)=max{D(i), D(j)+d(j)};
```

在图8-12中应用此计算方法, 可知应在节点 $r$ 、 $s$ 和 $v$ 处放置放大器 (如图8-13所示)。每个节点的 $D$ 值在节点内给出。

定理8-1 以上进程所需放大器数目最少。

证明 可通过对树的节点数 $n$ 进行归纳来证明。当 $n=1$ 时, 定理显然成立。设 $n=m$ 时, 定理成立, 其中 $m$ 为任意的自然数。设 $t$ 为有 $n+1$ 个节点的树。令 $X$ 为由上述进程所确定的放置放大器

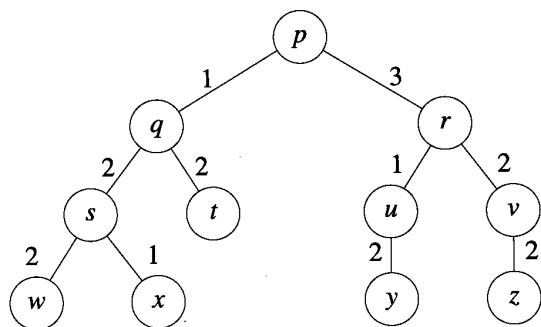


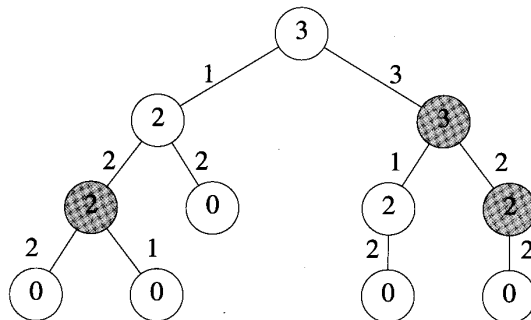
图8-12 树形分布网络



的节点的集合， $W$ 为满足容忍值限制的拥有最少放大器的节点集合，因此只需证 $|X| = |W|$ 。

若 $|X| = 0$ ，则 $|X| = |W|$ 。若 $|X| > 0$ ，则设 $z$ 为由上述进程给出的放置第一个放大器的节点， $t_z$ 为树 $t$ 中以 $z$ 为根的子树。因为 $D(z) + d(z) > \text{容忍值}$ ， $W$ 至少需包含 $t_z$ 中的某个节点 $u$ 。若 $W$ 中还包含了 $u$ 以外的元素，则 $W$ 一定不是最好的方案，因为通过在 $W - \{\text{所有的象 } u \text{ 这样的节点}\} + \{z\}$ 集合中放置放大器也能满足容忍值。因此 $W$ 中只包含节点 $u$ 。设 $W = W - \{u\}$ ， $t$

为从树 $t$ 中除去子树 $t_z$ （但保留 $z$ ）而得到的树，则对于 $t$ 而言， $W$ 为满足容忍值的放置放大器数目最少的方案。而且， $X = X - \{z\}$ 在树 $t$ 也满足容忍值。因 $t$ 中的节点数小于 $m+1$ ， $|X| = |W|$ ，因此 $|X| = |X| + 1 = |W| + 1 = |W|$ 。



信号调节器位于阴影节点  
节点内的数字为D值

图8-13 放置信号放大器的分布网络

当分布树中每个节点的孩子数都少于2时，可利用程序8-7给出的类BinaryTree和程序8-13给出的类Booster把该树描述成二叉树。域boost用来区分节点是否放置了放大器。二叉树的数据域类型为Booster。因为在8.9节中对程序8-7进行扩充时，定义了一个静态函数Output用来输出节点的数据域，因此必须重载输出操作符<<。

程序8-13 类Booster

```
class Booster {
    friend void main(void);
    friend void PlaceBoosters(BinaryTreeNode<Booster> *);
public:
    void Output(ostream& out) const
    {out << boost << ' ' << D << ' ' << d << ' ';}
private:
    int D,      // 叶节点的衰减量
        d;      // 父节点的衰减量
    bool boost; // 当且仅当本处设置放大器，则 boost为true
};

// 重载操作符 <<
ostream& operator<<(ostream& out, Booster x)
{x.Output(out); return out;}
```

可通过在二叉树中进行后序遍历来计算D值及确定放置放大器的节点集合。程序8-14用于访问每个节点。PlaceBoosters为Booster的一个友元，tolerance为一个全局变量。

程序8-14 在二叉树中放置放大器并计算D值

```
void PlaceBoosters(BinaryTreeNode<Booster> *x)
// 计算 *x.处的衰减量，如果衰减量超出了容忍值，则设置放大器
```

```

BinaryTreeNode<Booster> *y = x->LeftChild;
int degradation;
x->data.D = 0; // 初始化 x的衰减量
if (y) { // 从左孩子来计算
    degradation = y->data.D + y->data.d;
    if (degradation > tolerance)
        {y->data.boost = true;
        return;}
    else x->data.D = degradation;
}
y = x->RightChild;
if (y) { // 从右孩子来计算
    degradation = y->data.D + y->data.d;
    if (degradation > tolerance)
        {y->data.boost = true;
        else if (x->data.D < degradation)
            x->data.D = degradation;
        }
}
}

```

若X为类 BinaryTree<Booster>的成员，且d中的值为衰减值，boost域为0，则调用X.PostOrder(PlaceBoosters)可重新为D和boost赋值。PlaceBoosters的结果可通过X.PostOutput输出。由于PlaceBoosters的时间复杂性为 $\Theta(1)$ ，所以调用X.PostOrder(PlaceBoosters)所花费的时间为 $\Theta(n)$ ，其中n为树中的节点数。

#### 树的二叉树描述

当分布树t含有超过两个孩子的节点时，同样可以用二叉树来描述该树。此时，对于树t的每个节点x，可用其孩子节点的RightChild域把x的所有孩子链成一条链。x节点的LeftChild指针指向该链的第一个节点。x节点的RightChild域用来指向x的兄弟。

图8-14给出树及其二叉树描述。实线表示指向左孩子的指针，虚线表示指向其右孩子的指针。

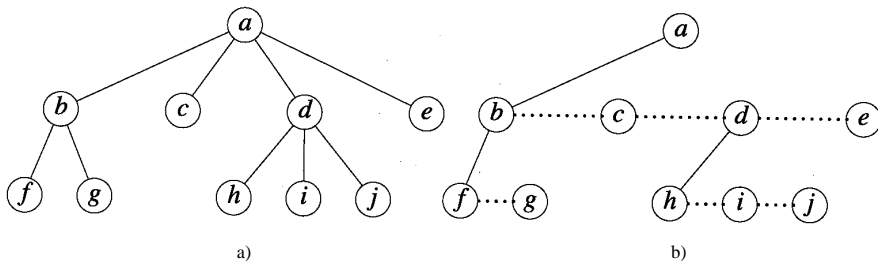


图8-14 树及其二叉树描述

a) 一棵树 b) 转换成二叉树

当一棵树用二叉树描述时，调用X.PostOrder(PlaceBoosters)不会产生预期的结果。在练习37中要求设计新的计算D和boost的函数。

#### 8.10.2 在线等价类

在3.8.3节中讨论了在线等价类问题。有n个元素从1到n编号，最开始，每一个元素在其自

己的类中，然后执行一系列 Find 和 Combine 操作。操作 Find( $e$ ) 返回元素  $e$  所在类的唯一特征，而 Combine( $a, b$ ) 用来合并包含  $a$  和  $b$  的类。在 3.8.3 节中，Combine( $a, b$ ) 是通过使用合并操作 Union( $i, j$ ) 完成的。其中  $i = \text{Find}(a)$ ,  $j = \text{Find}(b)$ 。3.8.3 节中的解决方案使用了链表，其复杂性为  $O(n + u \log u + f)$ ，其中  $u$  是合并操作的次数， $f$  是查找操作的次数。在线等价类问题也称作离散集合合并/搜索问题 (disjoint set union-find problem)。注意等价类可以看成元素的离散集合。

在这本节中，将采用另一种方案来解决在线等价类问题，其中把每个集合 (类) 描述为一棵树。图 8-15 给出一些描述成树的集合。而树中每个非根节点都指向其父节点。用根元素作为集合标识符。因此可以说元素 1、2、20、30 等在以 0 为根的集合中；元素 11、16、25、28 在以 16 为根的集合中；元素 15 在以 15 为根的集合中；元素 26 和 32 在以 26 为根的集合中 (或简称为集合 26)。

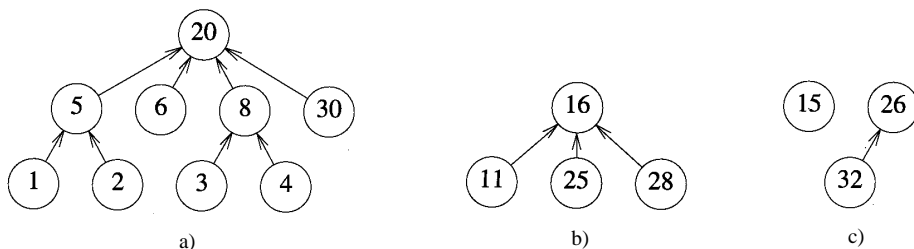


图8-15 离散集合的树形描述

### 1. 树形描述

合并/搜索问题的解决方案是使用模拟指针一个极好的例子。这里采用链表方法来描述树。每个节点必需有一个 parent 域，但不必有 children 域。要求能直接访问每个元素。为找到含有元素 10 的集合，需确定哪个节点表示元素 10，然后由其 parent 域一直搜索至根。若节点索引号为 1 到  $n$  且节点  $e$  表示元素  $e$ ，则很容易实现直接访问。每个 parent 域给出父节点的索引，因此 parent 域为整数类型。图 8-16 就是采用这种方法来描述图 8-15 的。节点内的数字是其 parent 域的值，节点外的数字为其索引。索引同时也就是该节点所表示的元素。根节点的 parent 域被置为 0。因为没有节点的索引会为 0，因此 parent=0 表示不指向任何节点 (即为空链)。

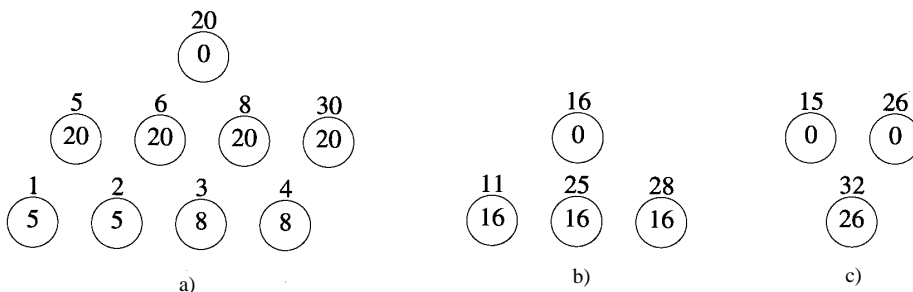


图8-16 图8-15中树的描述

因为每个节点只有一个域，只需如下定义：

```
int *parent;
```

### 2. 操作

开始时每个元素在仅包含它自己的一个集合中。为了创建初始结构,需分配数组 `parent` 并置 `parent[1:n]` 为 0。这由程序 8-15 中 `Initialize` 函数来完成。`Initialize` 的复杂性为  $\Theta(n)$ 。

程序 8-15 基于树结构的合并/搜索问题解决方案

```
void Initialize(int n)
{
    // 初始化, 每个类/树有一个元素
    parent = new int[n+1];
    for (int e = 1; e <= n; e++)
        parent[e] = 0;
}

int Find(int e)
{
    // 返回包含 e 的树的根节点
    while (parent[e])
        e = parent[e]; // 上移一层
    return e;
}

void Union(int i, int j)
{
    // 将根为 i 和 j 的两棵树进行合并
    parent[j] = i;
}
```

为找到包含元素  $e$  的集合, 从节点  $e$  出发, 由 `parent` 链搜索至根节点。若  $e=4$ , 集合的状态如图 8-15a 所示, 从 4 开始。由 `parent` 链到达节点 8。然后由节点 8 的 `parent` 链到达节点 20, 而节点 20 的 `parent` 为 0, 则 20 是根即该集合的标识符, 程序 8-15 中函数 `Find` 用来完成此功能。程序假设  $1 \leq e \leq n$  (也就说  $e$  为有效值)。Find 的复杂性为  $O(h)$ , 其中  $h$  为包含元素  $e$  的树的高度。

根为  $i$  和  $j$  ( $i \neq j$ ) 的集合的合并是通过把  $i$  作为  $j$  的子树, 或把  $j$  作为  $i$  的子树来实现的。例如, 取  $i=16$  且  $j=26$  (如图 8-15 所示), 若把  $i$  作为  $j$  的子树则结果为图 8-17a, 而当  $j$  为  $i$  的子树时, 结果为图 8-17b。程序 8-15 中的函数 `Union` 可用来执行合并操作, 假设在调用 `Union` 之前已检查了  $i \neq j$ 。通常把  $j$  作为  $i$  的子树来处理。`Union` 的时间复杂性为  $\Theta(1)$ 。

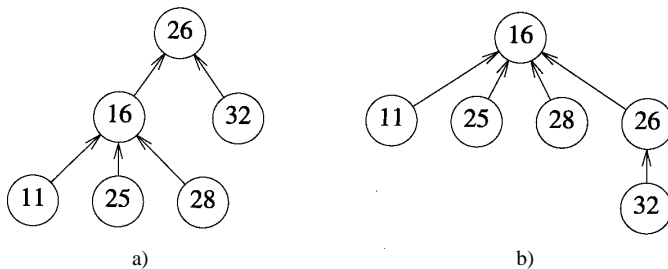


图 8-17 合并操作

### 3. 性能评价

假设要执行  $u$  次合并和  $f$  次查找。因为每次合并前都必须执行两次查找, 因此可假设  $f > u$ 。每次合并所需时间为  $\Theta(1)$ 。而每次查找所需时间由树的高度决定。在最坏情况下, 有  $m$  个元素的树的高度为  $m$ 。当执行以下操作序列时, 即可导致最坏情况出现:

$Union(2,1), Union(3,2), Union(4,3), Union(5,4), \dots$

因此每一次查找需花费  $\Theta(q)$  时间, 其中  $q$  是在执行查找之前所进行的合并操作的次数。

#### 4. 性能改进

在对树  $i$  和树  $j$  进行合并操作时, 可以通过使用重量规则或高度规则来提高合并 / 搜索算法的性能。

定义 [重量规则] 若树  $i$  节点数少于树  $j$  节点数, 则将  $j$  作为  $i$  的父节点。否则, 将  $i$  作为  $j$  的父节点。

定义 [高度规则] 若树  $i$  的高度小于树  $j$  的高度, 则将  $j$  作为  $i$  的父节点, 否则将  $i$  作为  $j$  的父节点。

若对图 8-15a 和 b 中两树进行合并, 则无论是采用重量规则还是高度规则, 根为 16 的树都将成为根为 20 的树的子树。若对图 8-18a 和 b 的树进行合并操作, 若按照重量规则, 根为 16 的树为根 20 树的子树。然而, 若按照高度规则, 根为 20 的树成为根为 16 的树的子树。

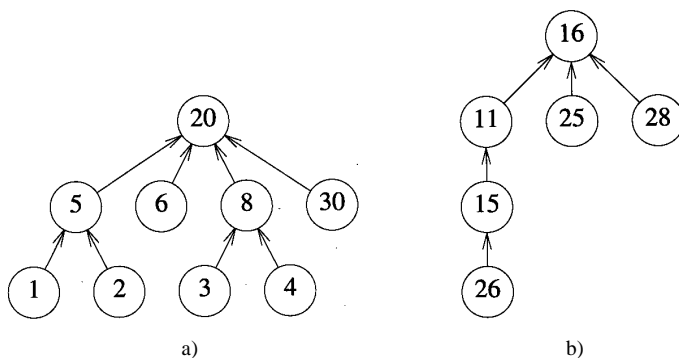


图8-18 两棵树

为了把重量规则应用到合并算法中, 在每个节点上增加一个布尔域 `root`。若当前节点为根节点则其 `root` 域为 `true`。每个根节点的 `parent` 域用来保存该树中节点的总数。对于图 8-15 的树, 当且仅当  $i=20, 16, 15$  或  $26$  时,  $root[i]=true$ 。并且当  $i=20, 16, 15$  和  $26$  时,  $parent[i]$  分别为 9、4、1 和 2。余下节点的 `parent` 域不变。

初始化、查找和合并的代码见程序 8-16, `root` 的定义如下:

```
bool *root;
```

练习 44 要求用一个数组来重新编写这三个函数, 每个数组元素包含两个域: `parent` 和 `root`。

虽然执行合并操作的时间有所增加, 但仍为一个常数, 即  $\Theta(1)$ 。定理 8-1 给出进行查找操作所需时间的最大值。

程序 8-16 用重量规则进合并

```
void Initialize(int n)
// 初始化, 每个类/树有一个元素
{
    root = new bool[n+1];
    parent = new int[n+1];
    for (int e = 1; e <= n; e++) {
        parent[e] = 1;
```

```

    root[e] = true;}
}

int Find(int e)
{// 返回包含 e的树的根节点
    while (!root[e])
        e = parent[e]; // 上移一层
    return e;
}

void Union(int i, int j)
{// 将根为 i 和 j的两棵树进行合并
    // 利用重量规则
    if (parent[i] < parent[j]) {
        // i 成为j的子树
        parent[j] += parent[i];
        root[i] = false;
        parent[i] = j; }
    else { // j 成为 i 的子树
        parent[i] += parent[j];
        root[j] = false;
        parent[j] = i;}
}

```

**定理8-2 [重量规则定理]** 假设从单元素集合出发，用重量规则进行合并操作(如程序8-16)。若按此方法构建有 $p$ 个节点的树 $t$ ，则 $t$ 的高度最多为  $\log_2 p + 1$ 。

**证明** 当 $p=1$ 时定理显然成立。假设当 $i < p-1$ 时，对所有具有 $i$ 个节点的树，定理均成立。下面将证明 $i=p$ 时定理也成立。设由程序8-16产生的树 $t$ 有 $p$ 个节点。最后一次合并操作为 $Union(k, j)$ ，设树 $j$ 的节点数为 $m$ ，树 $k$ 的节点数为 $p-m$ 。不失一般性可假设 $1 \leq m \leq p/2$ 。则树 $t$ 的高度与 $k$ 的高度要么相同，要么比 $j$ 的高度大1。若为前者，则 $t$ 的高度  $\log_2 (p-m) + 1 \leq \log_2 p + 1$ 。若后者为真则 $t$ 的高度  $\log_2 m + 2 \leq \log_2 p/2 + 2 \leq \log_2 p + 1$ 。

若从单元素集合出发，混合执行 $u$ 次合并和 $f$ 次查找序列，则每个集合不会超过 $u+1$ 个元素。由定理8-1知，若使用重量规则，合并和查找序列的代价(不包括初始化时间)为 $O(u + f \log u)$ 。若采用高度规则而非重量规则时，定理8-1的高度限制仍然适用。练习40，41和42给出了高度规则的具体应用。

在最坏情况下的性能可通过修改程序8-16的查找过程来提高，方法是缩短从元素 $e$ 到根的查找路径。路径的缩短可以通过称为路径压缩(path compression)的过程实现，其实现最少有3种不同方法。第一种方法，称为紧凑路径法(path compaction)，可改变从节点 $e$ (要查找的节点)到根的路径上所有节点的指针，使这些指针直接指向根节点。考察图8-19，若执行Find(10)操作，节点10、15和3位于从10到根的路径上，将其parent域改为2，就得到图8-20的树。(因为节点3本来已指向2，其parent域可不必修改。但在写程序时，为简化起见仍对其进行修改。)

虽然路径紧凑增加了单个查找操作的时间，但它减少了此后查找操作的时间。例如在图8-20的紧凑路径中查找元素10和15会更快。程序8-17给出紧凑规则的实现。

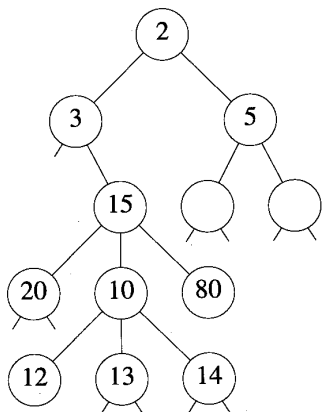


图8-19 样树

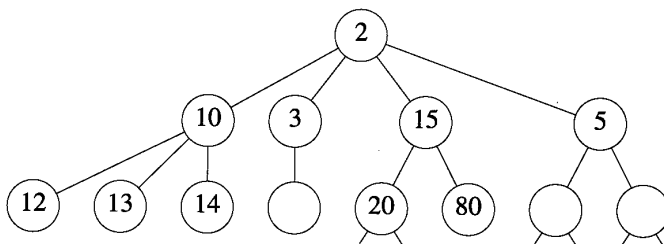


图8-20 路径紧凑

程序8-17 路径紧凑的实现代码

```

int Find(int e)
{
    // 返回包含 e 的树的根节点
    // 并对从 e 到根节点的路径进行紧凑
    int j = e;
    // 搜索根节点
    while (!root[j])
        j = parent[j];

    // 路径紧凑
    int f = e; // 从 e 开始
    while (f != j) { // f 不是根节点
        int pf = parent[f];
        parent[f] = j; // 上移至 2 层
        f = pf;        // f 移至原父节点
    }

    return j;
}

```

余下的两种路径压缩方法称为路径分割 (path splitting) 和路径对折 (path halving)。在路径分割中，改变从  $e$  到根节点路径上每个节点 (除了根和其子节点) 的 `parent` 指针，使其指向各自的祖父节点。在图 8-19 中，路径分割从节点 13 开始，结果得到图 8-21 中的树。在路径分割时，只考虑从  $e$  到根节点的一条路径就足够了。

在路径对折中，改变从  $e$  到根节点路径上每隔一个节点 (除了根和其子节点) 的 `parent` 域，使其指向各自的祖父节点。在路径对折中指针改变数仅为路径分割中的一半。同样，在路径对折中只考虑从  $e$  到根节点的一条路径就足够了。图 8-22 中给出图 8-19 从节点 13 开始进行路径对折的结果。

采用了合并和查找的性能改进算法，执行一串交错的合并和查找操作所需的时间几乎与合并和查找的次数成线性关系。为了更精确地计算时间复杂性，首先要定义 Ackermann 函数  $A(i, j)$  和其倒数  $(p, q)$ ：



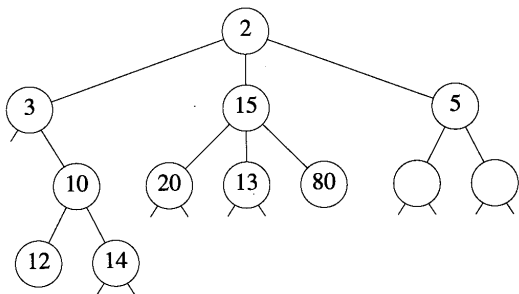


图8-21 路径分割

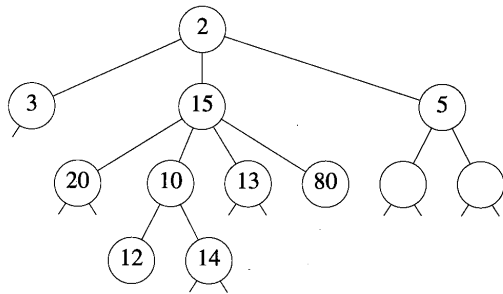


图8-22 路径对折

$$A(i, j) = 2j,$$

$$A(i, l) = A(i-1, 2)$$

$$A(i, j) = A(i-1, A(i, j-1))$$

$$(p, q) = \min\{z \mid a(z,$$

$$j-1$$

$$i-2$$

$$i, j-2$$

$$p/q) > \log_2 q\}, p, q-1$$

函数  $A(i, j)$  增长很快。相应地，随着  $p$  和  $q$  的增长而缓慢地增长。实际上，对于  $q < 2^{16} = 65536$  且  $p \leq q$ ，有  $A(3, 1) = 16$ ， $(p, q) \leq 3$ 。但  $A(4, 1)$  将是一个很大很大的数。在实际应用中， $q$  是集合中元素的个数， $p = n + f$  ( $f$  为查找的个数)，总是有  $(p, q) \leq 4$ 。

定理8-3 [Tarjan和Van Leeuwen] 设  $T(f, u)$  是交错的  $f$  次查找和  $u$  次合并操作所需的最大时间。假设  $u \leq n/2$ ，则

$$k_1(n + f(f + n, n)) \leq T(f, u) \leq k_2(n + f(f + n, n))$$

其中  $k_1$  和  $k_2$  为正常数。此定理适用于从单元素集合出发，采用重量或高度规则进行合并，同时按三种路径压缩方法的任意一种进行查找操作的情况。

定理8-3中，要求  $u \leq n/2$  并不是非常重要。因为当  $u < n/2$  时，某些元素在合并操作中并未涉及到。在合并和查找操作中，留在单元素集合中的元素，并不需要考虑，因为与这些元素有关的每一次操作可在  $O(1)$  时间内完成。虽然函数  $(f, u)$  增长很慢，但是合并/搜索的复杂性并不是与合并和查找的次数成线性关系。至于空间要求，每个元素需要一个节点即可。

## 练习

34. 画出图8-15a 和b，图8-17a 和b 和图8-18a 和b 的二叉树描述。
35. 画出图8-12的二叉树描述。(注意这类树的二叉树描述方法与用左孩子指针指向一个节点，右孩子指针指向另一节点的描述方法不同)。
36. 森林 (forest) 是一棵或多棵树的集合。在树的二叉树描述中，根没有右孩子。由此可以用二叉树来描述有  $m$  棵树的森林。首先得到森林中每棵树的二叉树描述，然后，第  $i$  棵作为第  $i-1$  棵树的右子树，画出图 8-15 中有4棵树的森林的二叉树描述，同时还有图 8-17和图8-18的二棵树的森林。
37. 设  $t$  为类 `BinanyTree` 的对象。假定  $t$  为分布树 (如图8-14所示) 的二叉树描述。给出计算  $t$  中每个节点的  $D$  和  $boost$  值的程序。程序应调用  $t.PostPrint()$  输出得到的结果，用适当的分布树来检查程序的正确性。
38. 设有  $n$  个集合，每个集合中元素各不相同

- 1) 证明若执行 $u$ 次合并操作, 则所有集合中的元素数都小于等于 $u+1$ 。
- 2) 证明在集合数目变成1之前, 最多执行了 $n-1$ 次合并操作。
- 3) 证明若执行的合并次数小于 $\lfloor n/2 \rfloor$ , 则至少有一个集合仅有一个元素。
- 4) 证明若执行了 $n$ 次合并操作, 则最少有 $\max\{n-2u, 0\}$ 个单元素集合。
39. 给出一个例子, 由单元素集合出发执行一系列合并操作, 使生成树的高度与定理 8-2给出的上限相等。假设每次合并均遵循重量规则。
40. 给出Union函数(见程序8-16)的另一个版本, 采用高度规则而不是重量规则。
41. 当采用高度规则而非重量规则时证明定理 8-2。
42. 给出一个例子, 由单元素集合出发执行一系列合并操作, 使生成树的高度等于定理 8-2给出的上限。假设每次合并均遵循高度规则。
43. 比较程序8-15和8-16的平均性能(用程序8-17的find函数代替程序8-16的find函数)。取 $n$ 的不同值时进行比较。对于每个 $n$ 值, 产生一随机序偶 $(i, j)$ 。用两个查找操作替换这个序偶(其中一个查找 $i$ , 另一个查找 $j$ )。若这2个元素在不同集合中, 则执行一次合并操作。使用多个不同的随机序偶重复进行实验。测试所有操作所需的总时间。自己设计实验的详细细节, 设计一个有意义的实验来比较两组程序的平均性能。写出实验过程和结果报告, 其中包括程序、平均时间表和图。
44. 重写程序8-16和8-17, 采用类型为Node的节点数组。Node可定义为类, 类中的每一实例均含有私有成员parent和root。若数组为 $E[0:n+1]$ , 则当且仅当 $e$ 为根节点时 $E[e].parent$ 是元素 $e$ 的父节点且 $E[e].root$ 为true。
45. 编写find函数, 采用路径分割而不是路径紧凑方法(程序8-17)。
46. 编写find函数, 采用路径对折而不是路径紧凑方法(程序8-17)。

## 8.11 参考及推荐读物

放置放大器的问题在下面论文中有深入研究: D.Paik, S.Reddy, S.Sahni. Deleting Vertices in Dags to Bound Path Lengths. *IEEE Transactions on Computers*, 43, 9, 1994, 1091~1096。还有 D.Paik, S.Reddy. Heuristics for the Placement of Flip-Flops in Partial Scan Designs and for the Placement of Signal Boosters in Lossy Circuits. *Sixth International Conference On VLSI Design*, 1993, 45~50。

在线等价类问题的树形描述在下面论文中有详细的介绍: R.Tarjan, J.Leeuwen. Worst Case Analysis of Set Union Algorithms. *Journal of the ACM*, 31, 2, 1984, 245~281。

China-pub.com

下载

## 第9章 优先队列

与第6章FIFO结构的队列不同，优先队列中元素出队列的顺序由元素的优先级决定。从优先队列中删除元素是根据优先权高或低的次序，而不是元素进入队列的次序。

可以利用堆数据结构来高效地实现优先队列。堆是一棵完全二叉树，可用 8.4节所介绍的公式化描述方法来高效存储完全二叉树。在高度和重量上取得平衡的左高树很适合于用来实现优先队列。本章的内容涵盖了堆和左高树。

在本章的应用部分，利用堆开发了一种复杂性为  $O(n \log n)$  的排序算法，称为堆排序。在第2章所介绍的对  $n$  个元素进行排序的算法，其复杂性均为  $O(n^2)$ 。虽然第3章介绍的箱子排序和基数排序算法的运行时间为  $\Theta(n)$ ，但算法中元素的取值必须在合适的范围内。堆排序是迄今为止所讨论的第一种复杂性优于  $O(n^2)$  的通用排序算法，第14章将讨论另一种与堆排序具有相同复杂性的排序算法。

从渐进复杂性的观点来看，堆排序是一种优化的排序算法，因为可以证明，任何通用的排序算法都是通过成对比较元素来获得  $(n \log n)$  复杂性的（见 14.4.2节）。

本节所考察的另外两个应用是机器调度和生成霍夫曼编码。机器调度问题属于 NP-复杂问题，对于这类问题不存在具有多项式时间复杂性的算法。而第2章提到的大量事实表明，只有具有多项式时间复杂性的算法才是可行的，因此，经常利用近似算法或启发式算法来解决 NP-完全问题，这些算法能在合理的时间内完成，但并不能保证找到最佳结果。对于机器调度应用，利用堆数据结构获得了有效解决机器调度问题的近似算法。本章没有提供有关左高树的应用，其实 6.4.4节中的工厂仿真在这方面是一个很好的应用。

### 9.1 引言

优先队列（priority queue）是0个或多个元素的集合，每个元素都有一个优先权或值，对优先队列执行的操作有 1) 查找；2) 插入一个新元素；3) 删除。在最小优先队列（min priority queue）中，查找操作用来搜索优先权最小的元素，删除操作用来删除该元素；对于最大优先队列（max priority queue），查找操作用来搜索优先权最大的元素，删除操作用来删除该元素。优先权队列中的元素可以有相同的优先权，查找与删除操作可根据任意优先权进行。

最大优先权队列的抽象数据类型描述如 ADT 9-1 所示，最小优先队列的抽象数据类型描述与之类似，只需将最大改为最小即可。

ADT 9-1 最大优先队列的抽象数据类型描述

---

抽象数据类型 *MaxPriorityQueue*{

实例

有限的元素集合，每个元素都有一个优先权

操作

*Create* ()：创建一个空的优先队列

*Size* ()：返回队列中的元素数目

*Max* ()：返回具有最大优先权的元素

*Insert* (*x*) : 将 *x* 插入队列

*DeleteMax* (*x*) : 从队列中删除具有最大优先权的元素, 并将该元素返回至 *x*

}

例9-1 假设我们对机器服务进行收费。每个用户每次使用机器所付费用都是相同的, 但每个用户所需要服务时间都不同。为获得最大利润, 假设只要有用户机器就不会空闲, 我们可以把等待使用该机器的用户组织成一个最小优先队列, 优先权即为用户所需服务时间。当一个新的用户需要使用机器时, 将他/她的请求加入优先队列。一旦机器可用, 则为需要最少服务时间 (即具有最高优先权) 的用户提供服务。

如果每个用户所需时间相同, 但用户愿意支付的费用不同, 则可以用支付费用作为优先权, 一旦机器可用, 所交费用最多的用户可最先得到服务, 这时就要选择最大优先队列。

例9-2 考察6.4.4节所介绍的工厂仿真问题, 对其事件队列所执行的操作有: 1) 查找具有最小完成时间的机器; 2) 改变该机器的完成时间。假设我们构造一个最小优先队列, 队列中的元素即为机器, 元素的优先权为该机器的完成时间。最小优先队列的查找操作可用来返回具有最小完成时间的机器。为了修改此机器的完成时间, 可以先从队列中删除具有最小优先权的元素, 然后用新的完成时间作为该元素的优先权并将其插入队列。实际上, 为了满足事件表的应用, 可以在最小优先队列的ADT表中新增加一个操作, 用来修改具有最小优先权元素的优先权。

最大优先队列也可用于工厂仿真问题。在6.4.4节中的仿真程序中, 每台机器按先进先出的方式来完成等待服务的任务, 因此可以为每台机器配置了一个FIFO队列。但如果将服务规则改为“一旦机器可用, 则从等待任务中选择优先权最大的任务进行处理”, 每台机器就需要一个最大优先队列。每台机器执行的操作有: 1) 每当一个新任务到达, 将其插入该机器的最大优先队列中; 2) 一旦机器可以开始运行一个新任务, 将具有最大优先权的任务从该机器的队列中删除, 并开始执行它。

当每个机器的服务规则如上述改变之后, 则需用一个最小优先队列来表示仿真问题中的事件表, 用一个最大优先队列来存储每台机器旁的等待任务。在6.4.4节的仿真模型中我们事先已经知道事件表的长度, 即机器的台数, 并且在整个仿真过程中事件表的长度不会改变, 所以可采用一个公式化描述的优先权队列来表示时间表。但更常见的是必须考虑加入新机器或移走旧机器的情况, 所以最好使用链表描述的优先队列, 这样在队列建立时就不必事先预测队列的最大长度, 也不必动态地改变数组的大小。

如同在6.4.4节中选择链表FIFO队列而不是公式化队列一样, 每个机器的优先权队列也最好是链表队列。每个机器的队列长度在仿真过程中不断变化, 而所有队列的长度之和却总是等于未完成任务数之和。

本章提供了优先权队列有效的描述方法。鉴于最大优先队列与最小优先队列十分类似, 故在此只明确给出了最大优先队列的描述。

## 9.2 线性表

描述最大优先队列最简单的方法是采用无序线性表。假设有一个具有  $n$  个元素的优先队列, 如果利用公式 (2-1), 那么插入操作可以十分容易地在表的右端末尾执行, 插入所需时间为  $\Theta(1)$ 。删除操作时必须查找优先权最大的元素, 即在未排序的  $n$  个元素中查找具有最大优先权的元素, 所以删除操作所需时间为  $\Theta(n)$ 。如果利用链表, 插入操作在链头执行, 时间为  $\Theta(1)$ ,

而每个删除操作所需时间为  $\Theta(n)$ 。

另一种描述方法是采用有序线性表，当使用公式 (2-1) 时元素按递增次序排列，使用链表时则按递减次序排列，这两种描述方法的删除时间均为  $\Theta(1)$ ，插入操作所需时间为  $\Theta(n)$ 。

## 练习

1. 利用公式化描述的无序线性表，设计一个 C++ 类来描述最大优先队列（即利用程序 3-1 中的 Linear List 类），使得插入时间为  $\Theta(1)$ ，对于  $n$  个元素的队列删除操作所需的最大时间为  $O(n)$ 。
2. 利用无序链表完成练习 1（即利用程序 3-8 中的类 Chain）。
3. 利用公式化描述的有序线性表重做练习 1，使得插入操作的时间为  $O(n)$ ，删除操作的最大时间为  $\Theta(1)$ 。
4. 利用程序 7-1 中的类 ShortedChain 重做练习 1。
5. 给出抽象数据类型 *MinPriorityQueue* 的描述，并采用公式化描述的无序线性表，用 C++ 类设计相应的最小优先队列。

## 9.3 堆

### 9.3.1 定义

定义 [最大树（最小树）] 每个节点的值都大于（小于）或等于其子节点（如果有的话）值的树。

最大树（max tree）与最小树（min tree）的例子分别如图 9-1、9-2 所示，虽然这些树都是二叉树，但最大树不必是二叉树，最大树或最小树节点的子节点个数可以大于 2。

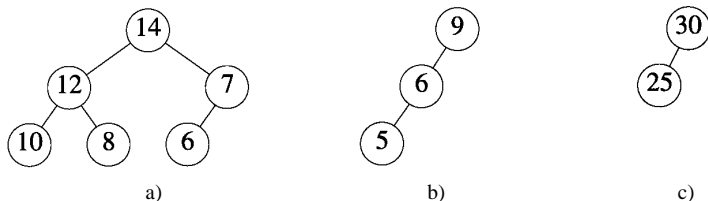


图9-1 最大树

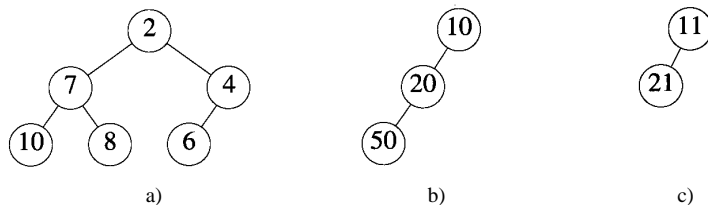


图9-2 最小树

定义 [最大堆（最小堆）] 最大（最小）的完全二叉树。

图 9-1b 所示的最大树并不是最大堆（max heap），另外两个最大树是最大堆。图 9-2b 所示的最小树不是最小堆（min heap），而另外两个是。

由于堆是完全二叉树，利用 8.4 节所介绍的公式化描述方案，可用一维数组有效地描述堆，利用二叉树的特性 4（见 8.3 节）可将堆中的节点移到它的父节点或它的一个子节点处。在后面的讨论中将用节点在数组中的位置来指定堆中的节点，如根的位置为 1，其左孩子为 2，右孩子为 3，等等。另外，注意到堆是完全二叉树，拥有  $n$  个元素的堆其高度为  $\lceil \log_2(n+1) \rceil$ ，因此，如果可在  $O(\text{height})$  时间内完成插入和删除操作，则这些操作的复杂性为  $O(\log_2 n)$ 。

### 9.3.2 最大堆的插入

图 9-3a 给出了一个具有 5 个元素的最大堆。由于堆是完全二叉树，当加入一个元素形成 6 元素堆时，其结构必如 9-3b 所示。如果插入元素的值为 1，则插入后该元素成为 2 的左孩子，相反，若新元素的值为 5，则该元素不能成为 2 的左孩子（否则将改变最大树的特性），应把 2 下移为左孩子（如图 9-3c 所示），同时还得决定在最大堆中 5 是否占据 2 原来的位置。由于父元素 20 大于等于新插入的元素 5，因此可以在原 2 所在位置插入新的元素。假设新元素的值为 21 而不是 5，这时，同图 9-3c 一样，把 2 下移为左孩子，由于 21 比父元素值大，所以 21 不能插入原来 2 所在位置，因此把 20 移到它的右孩子所在位置，21 插入堆的根节点（如图 9-3d 所示）。

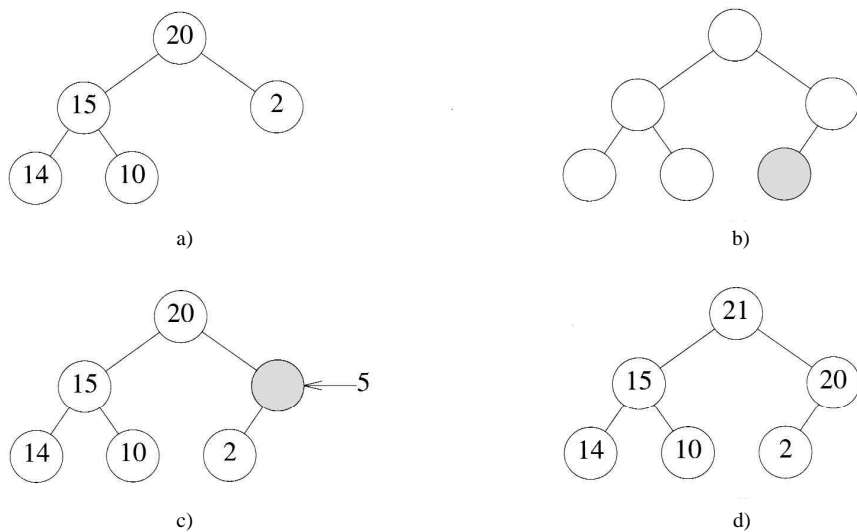


图9-3 最大堆的插入

插入策略从叶到根只有单一路径，每一层的工作需耗时  $\Theta(1)$ ，因此实现此策略的时间复杂性为  $O(\text{height}) = O(\log_2 n)$

### 9.3.3 最大堆的删除

从最大堆中删除一个元素时，该元素从堆的根部移出。例如，对图 9-3d 的最大堆进行删除操作即是移去元素 21，因此最大堆只剩下五个元素。此时，图 9-3d 中的二叉树需要重新构造，以便仍然为完全二叉树。为此可以移动位置 6 中的元素，即 2。这样就得到了正确的结构（如图 9-4a 所示），但此时根节点为空且元素 2 不在堆中，如果 2 直接插入根节点，得到的二叉树不是



最大树，根节点的元素应为 2、根的左孩子、根的右孩子三者中的最大值。这个值是 20，它被移到根节点，因此在位置 3 形成一个空位，由于这个位置没有孩子节点，2 可以插入，最后形成的最大堆如图 9-3a 所示。

现在假设要删除 20，在删除之后，堆的二叉树结构如图 9-4b 所示，为得到这个结构，10 从位置 5 移出，如果将 10 放在根节点，结果并不是最大堆。把根节点的两个孩子（15 和 2）中较大的一个移到根节点。假设将 10 插入位置 2，结果仍不是最大堆。因此将 14 上移，10 插入到位置 4，最后结果如图 9-4c 所示。

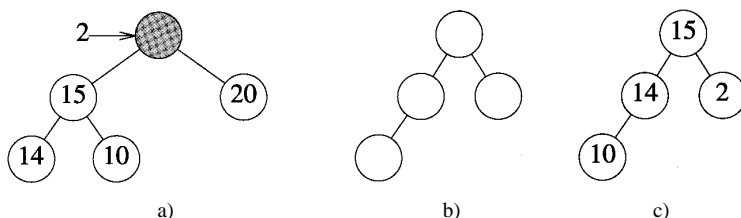


图9-4 最大堆的删除

删除策略产生了一条从堆的根节点到叶节点的单一路径，每层工作需耗时  $\Theta(1)$ ，因此实现此策略的时间复杂性为  $O(\text{height}) = O(\log_2 n)$

### 9.3.4 最大堆的初始化

在最大堆的几种应用中，包括 6.4.4 节中工厂仿真问题的事件表，开始时堆中已经含有  $n$  ( $n > 0$ ) 个元素。我们可以通过在初始为空的堆中执行  $n$  次插入操作来构建非空的堆，插入操作所需总时间为  $O(n \log n)$ ，也可利用不同的策略在  $\Theta(n)$  时间内完成堆的初始化。

假设开始数组  $a$  中有  $n$  个元素，另有  $n=10$ ， $a[1:10]$  中元素的关键值为  $[20, 12, 35, 15, 10, 80, 30, 17, 2, 1]$ ，这个数组可以用来表示如图 9-5a 所示的完全二叉树，这棵完全二叉树不是最大树。

为了将图 9-5a 的完全二叉树转化为最大堆，从第一个具有孩子的节点开始（即节点 10），这个元素在数组中的位置为  $i = \lfloor n/2 \rfloor$ ，如果以这个元素为根的子树已是最大堆，则此时不需调整，否则必须调整子树使之成为堆。随后，继续检查以  $i-1$ 、 $i-2$  等节点为根的子树，直到检查到整个二叉树的根节点（其位置为 1）。

下面对图 9-5a 中的二叉树完成这一系列工作。最初， $i=5$ ，由于  $10 > 1$ ，所以以位置  $i$  为根的子树已是最大堆。下一步，检查根节点在位置 4 的子树，由于  $15 < 17$ ，因此它不是最大堆，为将其变为最大堆，可将 15 与 17 进行交换，得到的树如图 9-5b 所示。然后检查以位置 3 为根的子树，为使其变为最大堆，将 80 与 35 进行交换。之后，检查根位于位置 2 的子树，通过重建过程使该子树成为最大堆。将该子树重构为最大堆时需确定孩子中较大的一个，因为  $12 < 17$ ，所以 17 成为重构子树的根，下一步将 12 与位置 4 的两个孩子中较大的一个进行比较，由于  $12 < 15$ ，15 被移到位置 4，空位 8 没有孩子，将 12 插入位置 8，形成的二叉树如图 9-5c。最后，检查位置 1，这时以位置 2 或位置 3 为根的子树已是最大堆了，然而  $20 < (\max[17, 80])$ ，因此 80 成为最大堆的根，当 80 移入根，位置 3 空出。由于  $20 < (\max[35, 30])$ ，位置 3 被 35 占据，最后 20 占据位置 6。图 9-5d 显示了最终形成的最大堆。

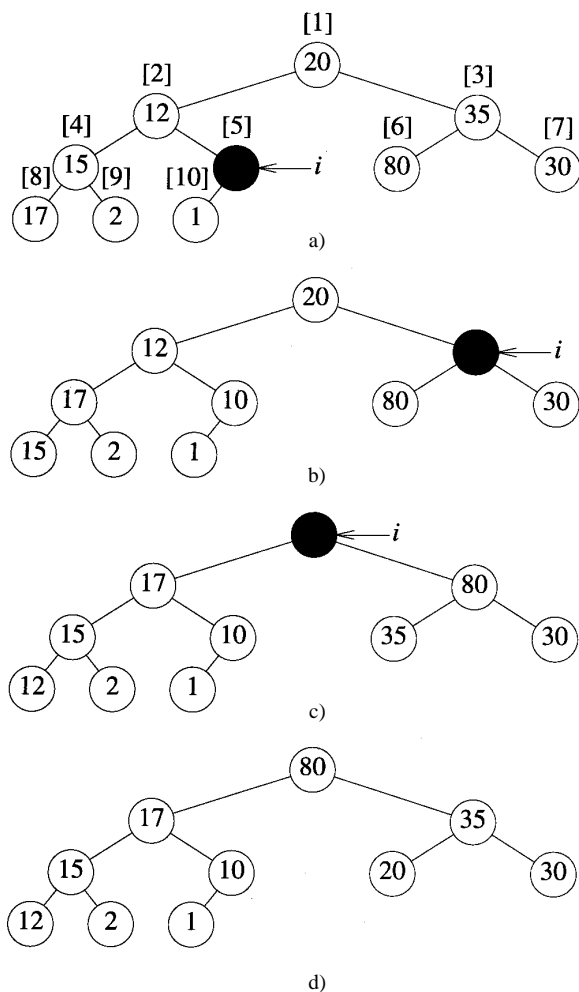


图9-5 最大堆的初始化

### 9.3.5 类MaxHeap

程序9-1给出了最大堆的类定义。 $n$  是私有成员，代表目前堆中元素的个数； $MaxSize$ 是堆的最大容量； $heap$ 为存贮堆元素的数组，省缺堆的大小为10个元素。

程序9-1 类MaxHeap

```
template<class T>
class MaxHeap {
public:
    MaxHeap(int MaxHeapSize = 10);
    ~MaxHeap() {delete [] heap;}
    int Size() const {return CurrentSize;}
    T Max() {if (CurrentSize == 0) throw OutOfBounds();
            return heap[1];}
    MaxHeap<T>& Insert(const T& x);
```

```
MaxHeap<T>& DeleteMax(T& x);  
void Initialize(T a[], int size, int ArraySize);  
private:  
    int CurrentSize, MaxSize;  
    T *heap; // 元素数组  
};
```

堆的构造函数见程序 9-2，构造函数只是简单地开辟一个足够大的数组使之能够存贮当元素个数达到最大值时的所有元素，它并不处理由 new 引发的 NoMem 异常。析构函数将删除此数组。size 函数尤其简单，仅仅返回 CurrentSize 的值。另一个简单的函数是 Max，如果最大堆为空，它将引发 OutOf Bounds 异常，如果不为空，则返回最大树的根的值。

程序 9-2 MaxHeap 的构造函数

```
template<class T>  
MaxHeap<T>::MaxHeap(int MaxHeapSize)  
{// 构造函数  
    MaxSize = MaxHeapSize;  
    heap = new T[MaxSize+1];  
    CurrentSize = 0;  
}
```

在 Insert（见程序 9-3）和 DeleteMax（见程序 9-4）的代码中假设已重载了关系操作符 <、> 和 >=。这些代码反映了 9.3.2 与 9.3.3 节所讨论内容。

程序 9-3 最大堆的插入

```
template<class T>  
MaxHeap<T>& MaxHeap<T>::Insert(const T& x)  
{// 把 x 插入到最大堆中  
    if (CurrentSize == MaxSize)  
        throw NoMem(); // 没有足够空间  
  
    // 为 x 寻找应插入位置  
    // i 从新的叶节点开始，并沿着树上升  
    int i = ++CurrentSize;  
    while (i != 1 && x > heap[i/2]) {  
        // 不能够把 x 放入 heap[i]  
        heap[i] = heap[i/2]; // 将元素下移  
        i /= 2; // 移向父节点  
    }  
  
    heap[i] = x;  
    return *this;  
}
```

在插入代码中，i 从新建的叶节点位置 CurrentSize 开始，对从该位置到根的路径进行遍历。对于每个位置 i，都要检查是否到达根（i=1）或在 i 处插入新元素不会改变最大树的性质（x.key > heap[i/2].key）。只要这两个条件中有一个满足，就可以在 i 处插入 x，否则，将执行 while 循环体，把位于 i/2 处的元素移到 i 处并把 i 处元素移到父节点（i/2）。对于一个具有 n 个元

素的最大堆（即  $\text{CurrentSize}=n$ ），while 循环的执行次数为  $O(\text{height})=O(\log n)$ ，且每次执行所需时间为  $\Theta(1)$ ，因此 Insert 的时间复杂性为  $O(\log n)$ 。

程序9-4 最大堆的删除

```
template<class T>
MaxHeap<T>& MaxHeap<T>::DeleteMax(T& x)
{// 将最大元素放入 x，并从堆中删除最大元素
    // 检查堆是否为空
    if (CurrentSize == 0)
        throw OutOfBounds(); // 队列空

    x = heap[1]; // 最大元素

    // 重构堆
    T y = heap[CurrentSize--]; // 最后一个元素

    // 从根开始，为 y 寻找合适的位置
    int i = 1, // 堆的当前节点
        ci = 2; // i 的孩子
    while (ci <= CurrentSize) {
        // heap[ci] 应是 i 的较大的孩子
        if (ci < CurrentSize &&
            heap[ci] < heap[ci+1]) ci++;

        // 能把 y 放入 heap[i] 吗？
        if (y >= heap[ci]) break; // 能

        // 不能
        heap[i] = heap[ci]; // 将孩子上移
        i = ci;              // 下移一层
        ci *= 2;
    }
    heap[i] = y;

    return *this;
}
```

在 DeleteMax 操作中，堆的根（即最大元素） $\text{heap}[1]$  被保存到变量  $x$  中，堆的最后一个元素  $\text{heap}[\text{CurrentSize}]$  被保存到变量  $y$  中，堆的大小（ $\text{CurrentSize}$ ）被减 1。在 while 循环中，开始查找一个合适的位置以便重新将  $y$  插入。从根部开始沿堆向下查找，对于具有  $n$  个元素的堆，while 循环的执行次数为  $O(\log n)$ ，且每次执行所花时间为  $\Theta(1)$ ，因此，DeleteMax 操作总的时间复杂性为  $O(\log n)$ 。注意到即使堆的元素个数为 0，代码也能正确执行，在这种情况下不执行 While 循环，对堆的位置 1 进行赋值是多余的。

Initialize 函数（见程序 9-5）使用数组  $a$  中的元素对最大堆进行初始化。初始时删除私有成员  $\text{heap}$  当前所指的数组，并使  $\text{heap}$  指向  $a[0]$ 。size 为数组  $a$  中的元素个数，ArraySize 是假设从  $a[1]$  算起数组  $a$  中所能容纳的最大元素个数。程序 9-5 的最初 4 行代码重新设置了最大堆的私有成员，使数组  $a$  代替数组  $\text{heap}$ 。在 for 循环中，从数组  $\text{heap}$ （即数组  $a$ ）的二叉树表示中最后一

个具有一个孩子的节点开始进行初始化，直至到达根节点。对于每个位置  $i$ ，在 while 循环中都保证根节点为  $i$  的子树已是最大堆。请注意 for 循环体与 DeleteMax（见程序 9-4）代码的相似性。

程序 9-5 初始化一个非空最大堆

```
template<class T>
void MaxHeap<T>::Initialize(T a[], int size, int ArraySize)
{// 把最大堆初始化为数组 a.
    delete [] heap;
    heap = a;
    CurrentSize = size;
    MaxSize = ArraySize;

    // 产生一个最大堆
    for (int i = CurrentSize/2; i >= 1; i--) {
        T y = heap[i]; // 子树的根

        // 寻找放置 y 的位置
        int c = 2*i; // c 的父节点是 y 的目标位置
        while (c <= CurrentSize) {
            // heap[c] 应是较大的同胞节点
            if (c < CurrentSize &&
                heap[c] < heap[c+1]) c++;

            // 能把 y 放入 heap[c/2] 吗?
            if (y >= heap[c]) break; // 能

            // 不能
            heap[c/2] = heap[c]; // 将孩子上移
            c *= 2; // 下移一层
        }
        heap[c/2] = y;
    }
}
```

在 Initialize 中将整个数组的元素放入一个类中，这样一来，当超出了最大堆的范围而又仍想访问数组  $a$  中的元素时将会产生问题。为避免这个问题，在 MaxHeap 类的定义中加入一个共享成员函数 Deactivate：

```
void Deactivate() {heap=0;}
```

通过使用该函数，可以防止在调用堆的析构函数时将数组  $a$  删除。

Initialize 函数的复杂性

如果元素个数为  $n$ ，Initialize 函数（见程序 9-5）中 for 循环每次所花时间为  $O(\log n)$ ，循环次数为  $n/2$ ，所以总的复杂性为  $O(n \log n)$ 。注意符号  $O$  代表算法复杂性的上限。实际应用中，Initialize 的复杂性要比  $O(n \log n)$  好一些。经过更仔细的分析，我们发现其真正的复杂性为  $\Theta(n)$ 。

Initialize 的 while 循环每次所花费时间为  $O(h_i)$ ，其中  $h_i$  是以  $i$  位置为根节点的子树的高度。完全二叉树  $a[1:n]$  的高度为  $h = \lceil \log_2(n+1) \rceil$ 。在第  $j$  层最多含有  $2^{j-1}$  个节点，因此最多有  $2^{j-1}$  个节点具有相同的高度  $h_i = h - j + 1$ ，所以最大堆的初始化时间为：

$$O\left(\sum_{j=1}^{h-1} 2^{j-1} (h-j+1)\right) = O\left(\sum_{k=1}^{h-1} k 2^{h-k}\right) = O\left(2^h \sum_{k=1}^{h-1} (k/2^k)\right) = O(2^h) = O(n)$$

由于for 循环执行了  $n/2$  次，其复杂性为  $O(n)$ 。将两者综合考虑，可以得到 Initialize 的复杂性为  $\Theta(n)$ 。

## 练习

6. 在MaxHeap类中加入共享成员函数 IsEmpty和IsFull，前者当且仅当最大堆为空时返回 true。后者当且仅当最大堆已满时返回 true。

7. 模仿最大堆的类定义构造一个最小堆的定义，并测试代码的正确性。

8. 在MaxHeap类中加入一个共享成员函数 ChangeMax(x)，将当前最大元素改为元素 x，x 的值可以大于或小于当前最大元素的值，与删除最大元素的操作一样，代码沿着从根开始的一条向下的路径遍历。执行过程中当最大堆为空时，引发一个 OutOfBounds异常，代码的复杂性应为  $O(\log n)$ ，其中  $n$  是最大堆中元素个数，证明这个结论。

9. 由于在删除操作（见程序 9-4）中重新插入最大堆的元素  $y$  是从堆的底部移出的那个元素，我们期望它仍插在接近底部的地方。重新写一个 DeleteMax函数，让根节点的空位置先移到叶节点，然后  $y$  通过这个叶节点往上找到它的合适位置。通过实验比较新代码是否比旧代码执行速度快。

10. 根据假设：

1) 在创建堆时，提供两个元素 MaxElement和MinElement，堆中没有元素比MaxElement大，也没有元素比MinElement小。

2) 一个具有  $n$  个元素的堆需要一个数组 heap[0:2n+1]。

3)  $n$  个元素，如本节所描述的那样存贮在 heap[1:n]中。

4) MaxElement存贮在 heap[0]中。

5) MinElement存贮在 heap[n+1:2n+1]中。

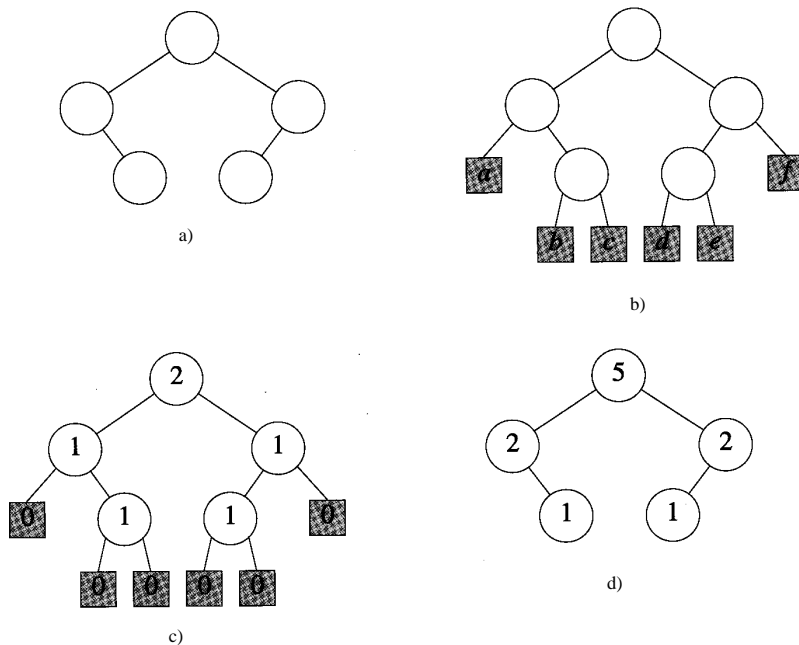
重写MaxHeap中的成员函数。这些假设可使 Insert和Delete操作得以简化。将本练习与本节中的实现作一对比。

## 9.4 左高树

### 9.4.1 高度与宽度优先的最大及最小左高树

9.3节的堆结构是一种隐式数据结构（implicit data structure），用完全二叉树表示的堆在数组中是隐式存贮的（即没有明确的指针或其他数据能够重构这种结构）。由于没有存贮结构信息，这种描述方法空间利用率很高，事实上没有空间浪费。尽管堆结构的时间和空间效率都很高，但它不适用于所有优先队列的应用，尤其是当需要合并两个优先队列或多个长度不同的队列时。因此需要借助于其他数据结构来实现这类应用，左高树（leftist tree）就能满足这种要求。

考察一棵二叉树，它有一类特殊的节点叫做外部节点（external node），用来代替树中的空子树，其余节点叫做内部节点（internal node）。增加了外部节点的二叉树被称为扩充二叉树（extended binary tree），图9-6a 给出了一棵二叉树，其相应的扩充二叉树如图 9-6b 所示，外部节点用阴影框表示，为了方便起见，这些节点用  $a \sim f$  标注。

图9-6  $s$  和  $w$  的值a) 一棵二叉树 b) 扩充二叉树 c)  $s$  的值 d)  $w$  的值

令  $s(x)$  为从节点  $x$  到它的子树的外部节点的所有路径中最短的一条, 根据  $s(x)$  的定义可知, 若  $x$  是外部节点, 则  $s$  的值为 0, 若  $x$  为内部节点, 则它的  $s$  值是:

$$\min\{s(L), s(R)\} + 1$$

其中  $L$  与  $R$  分别为  $x$  的左右孩子。扩充二叉树 (如图 9-6b 所示) 各节点的  $s$  值如图 9-6c 所示。

**定义 [高度优先左高树]** 当且仅当一棵二叉树的任何一个内部节点, 其左孩子的  $s$  值大于等于右孩子的  $s$  值时, 该二叉树为高度优先左高树 (height-biased leftist tree, HBLT)。

图 9-6a 所示的二叉树不是 HBLT。考察外部节点  $a$  的父节点, 它的左孩子的  $s$  值为 0, 而右孩子的为 1, 所有其他内部节点均满足 HBLT 的定义。因此, 若将图 9-6a 中节点  $a$  的父节点的左右子树进行交换, 所得到的二叉树即为 HBLT。

**定理 9-1** 令  $x$  为一个 HBLT 的内部节点, 则

- 1) 以  $x$  为根的子树的节点数目至少为  $2^{s(x)} - 1$ 。
- 2) 若子树  $x$  有  $m$  个节点,  $s(x)$  最多为  $\log_2(m+1)$ 。
- 3) 通过最右路径 (即, 此路径是从  $x$  开始沿右孩子移动) 从  $x$  到达外部节点的路径长度为  $s(x)$ 。

**证明** 根据  $s(x)$  的定义可知, 从  $x$  节点往下第  $s(x)-1$  层没有外部节点 (否则  $x$  的  $s$  值将更小)。以  $x$  为根的子树在当前层只有一个节点  $x$ , 下一层有两个, 再下一层有四个, ...,  $x$  层往下第  $s(x)-1$  层有个  $2^{s(x)-1}$ , 在  $s(x)-1$  层以下可能还有其他节点, 因此子树  $x$  的节点数目至少为  $\sum_{i=0}^{s(x)-1} 2^i = 2^{s(x)} - 1$ 。从 1) 可以推出 2)。根据  $s$  的定义以及 HBLT 一个节点的左孩子的  $s$  值总是大于等于其右孩子的  $s$  值, 可以推得 3) 成立。



定义 [最大HBLT] 即同时又是最大树的HBLT; [最小HBLT] 即同时又是最小树的HBLT。

图9-1的最大树及图9-2的最小树都是HBLT。因此, 图9-1的树是最大HBLT, 图9-2中的树是最小HBLT。最大优先队列可以用最大HBLT表示, 最小优先队列可用最小HBLT表示。

可以通过考察子树的节点数目而不是从根到外部节点的路径来得到另一类左高树。定义  $x$  的重量  $w(x)$  为以  $x$  为根的子树的内部节点数目。注意到若  $x$  是外部节点, 则其重量为 0; 若  $x$  为内部节点, 其重量为其孩子节点的重量的和加 1, 图9-6a 中二叉树各节点的重量的图 9-6d 所示。

定义 [重量优先左高树] 当且仅当一棵二叉树的任何一个内部节点, 其左孩子的  $w$  值大于等于右孩子的  $w$  值时, 该二叉树为重量优先左高树 (weight-biased leftist tree, WBLT); [最大 (小) WBLT] 即同时又是最大 (小) 树的 WBLT。

同HBLT类似, 具有  $m$  个节点的 WBLT 的最右路径长度最多为  $\log_2(m+1)$ 。可以对 WBLT 与 HBLT 执行优先队列的查找、插入、删除操作, 其时间复杂性与堆的相应操作相同。同堆一样, WBLT 与 HBLT 可在线性时间内完成初始化。用 WBLT 或 HBLT 描述的两个优先队列可在对数时间内合并为一个, 而当用堆描述优先队列时, 则不能在对数时间内完成合并。

WBLT 中的查找、插入、删除、合并和初始化操作与 HBLT 中的相应操作很相似, 因此, 以下仅介绍有关 HBLT 的操作。WBLT 的操作将留做练习 (练习 12)。

#### 9.4.2 最大HBLT的插入

最大HBLT的插入操作可借助于最大HBLT的合并操作来完成。假设将元素  $x$  插入到名为  $H$  的最大HBLT中, 如果建造一棵仅有一个元素  $x$  的最大HBLT然后将它与  $H$  进行合并, 结果得到的最大HBLT将包括  $H$  中的全部元素及元素  $x$ 。因此插入操作只需先建立一棵仅包含欲插入元素的HBLT, 然后将它与原来的HBLT合并即可。

#### 9.4.3 最大HBLT的删除

根是最大元素, 如果根被删除, 将留下分别以其左右孩子为根的两棵 HBLT 的子树。将这两棵最大HBLT合并到一起, 便得到包含除删除元素外所有元素的最大 HBLT, 所以删除操作可以通过删除根元素并对两个子树进行合并来实现。

#### 9.4.4 合并两棵最大HBLT

具有  $n$  个元素的最大 HBLT, 其最右路径的长度为  $O(\log n)$ 。合并操作仅需遍历欲合并的 HBLT 的最右路径。由于在两条最右路径的每个节点上只需耗时  $O(1)$ , 因此将两棵 HBLT 进行合并具有对数复杂性。通过以上观察, 在我们所设计的合并算法中, 仅需移动右孩子。

合并策略最好用递归来实现。令  $A$ 、 $B$  为需要合并的两棵最大 HBLT, 如果其中一个为空, 则将另一个作为合并的结果, 因此可以假设两者均不为空。为实现合并, 先比较两个根元素, 较大者作为合并后的 HBLT 的根。假定  $A$  具有较大的根, 且其左子树为  $L$ ,  $C$  是由  $A$  的右子树与  $B$  合并而成的 HBLT。  $A$  与  $B$  合并所得结果即是以  $A$  的根为根,  $L$  与  $C$  为左右子树的最大 HBLT。如果  $L$  的  $s$  值小于  $C$  的  $s$  值, 则  $C$  为左子树, 否则  $L$  为左子树。

例9-3 考察图9-7a 所示的两棵最大HBLT。每个节点的  $s$  值显示在节点的外侧, 元素的值标在

节点内部。图中总是将要合并的两棵最大 HBLT 中具有较大根的树画在左边。根据这种约定，左边 HBLT 的根总是最后所得到的 HBLT 的根。此外，右边 HBLT 的根用阴影来标出。

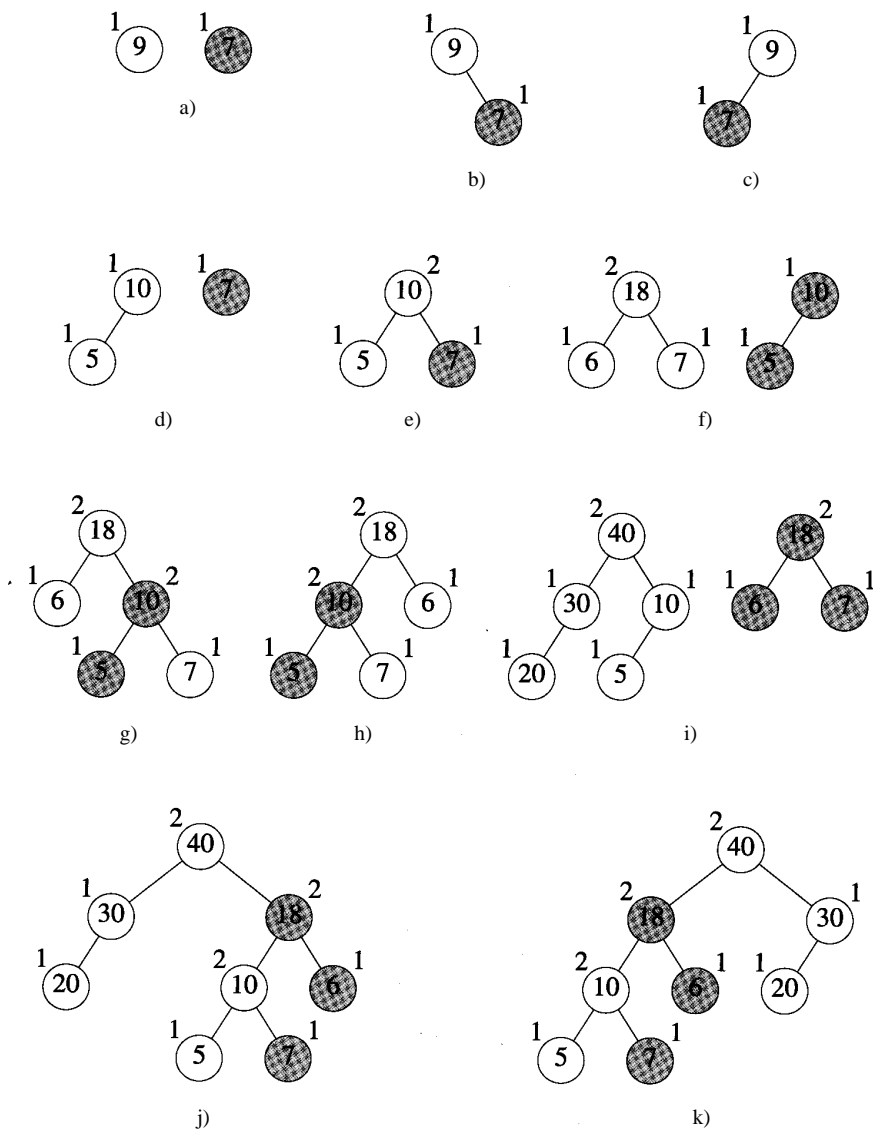


图9-7 最大HBLT的合并

因为9的右子树为空，因此将9的右子树与以7为根的HBLT进行合并，得到的仍是以7为根的树，把这棵树暂时作为9的右子树，可以得到如图9-7b所示的最大树。由于此时9的左子树s值为0而右子树s值为1，因此将其左右子树进行交换，得到如图9-7c所示的最大HBLT。

下面来考察合并图9-7d所示的两棵最大HBLT。毫无疑问，左子树的根将作为最终的根。当10的右子树与根为7的HBLT进行合并时，所得结果仍为后者。如果把这棵HBLT作为10的右子树，可得到图9-7e所示的最大树。比较10的左右孩子的s值，发现没有必要再进行交换。

现在考察合并图9-7f中的两棵最大HBLT。左子树的根无疑将作为最终的根。首先把18的

右子树与以10为根的最大HBLT进行合并，其过程与图9-7d中的情形完全一致，合并的结果为图9-7e所示的最大HBLT。把这棵树作为18的右子树，即可得到图9-7f所示的最大树。比较18的左右子树的 $s$ 值，可知这两棵树必须交换，交换后所得结果如图9-7h所示。

作为最后一个例子，来合并图9-7i中的两棵最大HBLT。同前面的例子一样，左子树的根将成为最终的根。首先把40的右子树与根为18的最大HBLT进行合并，这个过程与图9-7f中的合并过程完全一致，合并的结果为图9-7g所示的最大HBLT。把图9-7g中的最大HBLT作为40的右子树，即可得到图9-7j所示的最大树。由于40的左子树的 $s$ 值比其右子树的 $s$ 值小，因此将这两棵树进行交换，最后得到图9-7k所示的最大HBLT。注意到在合并图9-7i中的最大HBLT时，先移动到40的右孩子，再移动到18的左孩子，最后移动到10的右孩子，因此所有移动都是按照当前最大HBLT的最右路径进行的。

#### 9.4.5 初始化最大HBLT

通过将 $n$ 个元素插入到最初为空的最大HBLT中来对其进行初始化，所需时间为 $O(\log n)$ 。为得到具有线性时间的初始化算法，首先创建 $n$ 个最大HBLT，每个树中仅包含 $n$ 个元素中的某一个，这 $n$ 棵树排成一个FIFO队列，然后从队列中依次删除两个HBLT，将其合并，然后再加入队列末尾，直到最后只有一棵HBLT。

例9-4 我们希望构造具有五个元素：7,1,9,11,2的一棵最大HBLT。为此，首先构造五个单元素的最大HBLT，并形成一个FIFO队列。把最前面的两个最大HBLT 7和1从队列中删除并进行合并，所得结果如图9-8a所示，将该结果加入到队列中。接下来从队列中删除两棵最大HBLT 9和11并进行合并，所得结果如图9-8b所示，也将该结果加入到队列中。现在继续从队列中删除最大HBLT 2及图9-8a所得到的HBLT，并进行合并，所得结果（如图9-8c所示）加入队列。下一对从队列中删除的最大HBLT如图9-8b与9-8c所示，经合并得到的最大HBLT如图9-8d所示，将该结果插入到队列中。至此，队列中只有一棵最大HBLT，初始化工作完成。

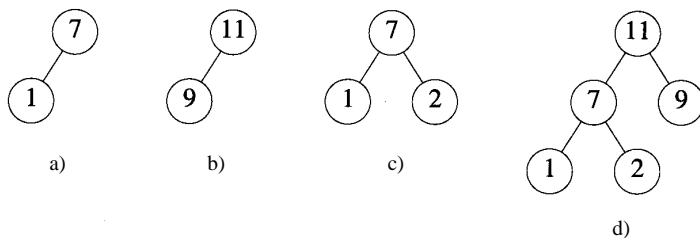


图9-8 最大HBLT的初始化

#### 9.4.6 类MaxHBLT

最大HBLT的每个节点均需要 data、LeftChild、RightChild和 $s$ 四个域，相应的节点类为HBLTNode（见程序9-6）。在HBLTNode的构造函数中要求提供 data和 $s$ ，同时将LeftChild和RightChild置为0。

程序9-6 HBLT的节点类

```
template <class T>
```

```
class HBLTNode {
    friend MaxHBLT<T>;
public:
    HBLTNode(const T& e, const int sh)
    {data = e;
     s = sh;
     LeftChild = RightChild = 0;}
private:
    int s; // 节点的s 值
    T data;
    HBLTNode<T> *LeftChild, *RightChild;
};
```

最大HBLT可用程序9-7中的MaxHBLT类来实现，MaxHBLT类的每个对象都有一个唯一的私有成员root，用来指向最大HBLT的根。构造函数在初始化时将root置为0，因此初始的最大HBLT为空。析构函数通过调用私有成员函数Free来删除HBLT中的所有节点。Free函数按后序遍历整棵HBLT，每访问一个节点就删除该节点。Free的代码与8.9节中二叉树的私有成员函数Free的代码相同。

程序9-7 MaxHBLT类

```
template<class T>
class MaxHBLT {
public:
    MaxHBLT() {root = 0;}
    ~MaxHBLT() {Free(root);}
    T Max() {if (!root) throw OutOfBounds();
             return root->data;}
    MaxHBLT<T>& Insert(const T& x);
    MaxHBLT<T>& DeleteMax(T& x);
    MaxHBLT<T>& Meld(MaxHBLT<T>& x) {
        Meld(root,x.root);
        x.root = 0;
        return *this;}
    void Initialize(T a[], int n);
private:
    void Free(HBLTNode<T> *t);
    void Meld(HBLTNode<T> * &x, HBLTNode<T> * y);
    HBLTNode<T> *root; // 指向树根的指针
};
```

由于插入、删除最大元素及初始化操作都需要使用合并操作，因此首先考察合并操作。共享成员函数Meld用于合并两棵最大HBLT，不过它是通过调用私有成员函数Meld来完成合并操作的。私有成员函数Meld用来实际完成合并任务，它带有两个参数x和y，二者分别指向欲合并的最大HBLT的根，合并所得到的HBLT的根保存在x中。在共享函数Meld中，当root与x.root合并完成之后，该函数将x的根域置为0（这是为了防止x的原节点被意外删除），并返回所得到的合并树的引用。

程序9-8给出了私有函数Meld。该函数首先处理要合并的树中至少有一个为空的特殊情况。当没有空树时要确保x指向根值较大的树，如果x不是指向根值较大的树，则将x与y的指针进行交换。接下来把x的右子树与以y为根的最大HBLT进行递归合并。合并后为保证整棵树为最大HBLT，x的左右孩子可能需要交换，这是通过计算x的s值来确定的。

程序9-8 合并两棵左高树

---

```
template<class T>
void MaxHBLT<T>::Meld(HBLTNode<T>* &x, HBLTNode<T>* y)
{
    // 合并两棵根分别为*x和*y的左高树
    // 返回指向新根 x的指针
    if (!y) return; // y 为空
    if (!x) // x为空
    {
        x = y;
        return;
    }

    // x和y 均为空
    if (x->data < y->data) Swap(x,y);
    // 现在 x->data >= y->data
    Meld(x->RightChild,y);
    if (!x->LeftChild) { // 左子树为空
        // 交换子树
        x->LeftChild = x->RightChild;
        x->RightChild = 0;
        x->s = 1;
    }
    else { // 检查是否需要交换子树
        if (x->LeftChild->s < x->RightChild->s)
            Swap(x->LeftChild,x->RightChild);
        x->s = x->RightChild->s + 1;
    }
}
```

---

为了将元素x插入到一棵最大HBLT中，程序9-9中的代码先产生一棵只有一个元素x的最大HBLT，然后通过私有成员函数Meld将此树与欲插入的最大HBLT进行合并。函数将返回一个指向所得最大HBLT的指针。该函数并不捕获可能由new产生的异常。

程序9-9 最大HBLT的插入

---

```
template<class T>
MaxHBLT<T>& MaxHBLT<T>::Insert(const T& x)
{
    // 把x插入到左高树中
    // 创建带有一个节点的树
    HBLTNode<T>* q = new HBLTNode<T>(x,1);
    // 把q与原树进行合并
    Meld(root,q);
    return *this;
}
```

---

在DeleteMax代码（见程序9-10）中，若最大HBLT为空，则引发OutOfBounds异常；若不为空，则将其左右子树的根分别保存在指针L与R中，然后将根删除，并把子树L和R合并。

程序9-10 从最大HBLT中删除最大元素

```
template<class T>
MaxHBLT<T>& MaxHBLT<T>::DeleteMax(T& x)
{
    // 删除最大元素，并将其放入 x
    if (!root) throw OutOfBounds();

    // 树不为空
    x = root->data; // 最大元素
    HBLTNode<T> *L = root->LeftChild;
    HBLTNode<T> *R = root->RightChild;
    delete root;
    root = L;
    Meld(root,R);
    return *this;
}
```

最大HBLT的初始化代码见程序9-11。代码中利用一个公式化 FIFO 队列来保存初始化过程中所产生的最大HBLT。在第一个for循环中，产生了n 个仅含一个元素的最大HBLT并依次加入最初为空的队列。在下一个for 循环中，每次从队列中删除两个最大HBLT，将其合并，并把结果加入队列中。当for 循环结束时，队列中仅含一棵有n 个元素的最大HBLT（假设 $n > 1$ ）。

程序9-11 最大HBLT的初始化

```
template<class T>
void MaxHBLT<T>::Initialize(T a[], int n)
{
    // 初始化有n个元素的 HBLT 树
    Queue<HBLTNode<T>*> Q(n);
    Free(root); // 删除老节点
    // 对树的队列进行初始化
    for (int i = 1; i <= n; i++) {
        // 创建只有一个节点的树
        HBLTNode<T> *q = new HBLTNode<T> (a[i],1);
        Q.Add(q);
    }

    // 不断合并队列中的树
    HBLTNode<T> *b, *c;
    for (i = 1; i <= n - 1; i++) {
        // 删除并合并两棵树
        Q.Delete(b).Delete(c);
        Meld(b,c);
        // 把合并后所得到的树放入队列
        Q.Add(b);
    }

    if (n) Q.Delete(root);
}
```

## 复杂性分析

构造函数需耗时  $\Theta(1)$ ，而析构函数需耗时  $\Theta(n)$ ，其中  $n$  为欲删除的最大 HBLT 中的元素个数。Max 函数的复杂性为  $\Theta(1)$ 。Insert、DeleteMax 及共享成员函数 Meld 的复杂性与私有成员函数 Meld 的复杂性相同。由于私有成员函数 Meld 仅在以  $*x$  和  $*y$  为根的树的右子树中移动，因此该函数的复杂性为  $O(x \rightarrow s + y \rightarrow s)$ 。由于  $*x$  与  $*y$  的最大  $s$  值分别为  $\log_2(m+1)$  与  $\log_2(n+1)$ ，其中  $m$  与  $n$  分别是以  $*x$  和  $*y$  为根的最大 HBLT 中的元素个数，所以私有成员函数 Meld 的复杂性为  $O(\log mn)$ 。

为了分析 Initialize 的复杂性，为简单起见，假设  $n$  是 2 的幂次方。首先合并  $n/2$  对具有一个元素的最大 HBLT，然后合并  $n/4$  个含有两个元素的最大 HBLT，继而合并  $n/8$  个含有 4 个元素的最大 HBLT，……。由于合并两棵含  $2^i$  个元素的最大 HBLT 需耗时  $O(i+1)$ ，因此 Initialize 所花费的总时间为：

$$O(n/2 + 2*(n/4) + 3*(n/8) + \cdots) = O(n \sum \frac{i}{2^i}) = O(n)$$

## 练习

11. 写出 MinHBLT 类的代码，该类与 MaxHBLT 类的差别仅在于原来最大 HBLT 中的类成员现在是最小 HBLT 的成员，用 Min 与 DeleteMin 操作来代替 Max 与 DeleteMax 操作。

12. 1) 图 9-1 中哪些（如果有）二叉树是 WBLT？

2) 令  $x$  为 WBLT 中的一个节点，对  $w(x)$  进行归纳，来证明从  $x$  出发到达一个外部节点的最右路径的最大长度为  $\log_2(w(x)+1)$ 。

3) 用 WBLTNode 类来描述重量优先左高树的节点，每个类成员具有  $w$ （重量）、data、LeftChild 和 RightChild 域。

4) 设计类 MaxWBLT 来描述最大 WBLT。类中应该包括 Max、Insert、DeleteMax、Meld 及 Initialize 函数，这些函数分别对应于最大 HBLT 中的相应函数。代码中每个函数应与 MaxHBLT 中的相应函数具有相同的复杂性。用非递归的代码实现私有成员函数 Meld。注意由于每个节点的  $w$  值可在向下移动的过程中计算出来，因此在使用 WBLT 时，自底向上的递归展开过程（见程序 9-8）是多余的，但若使用 HBLT 则是必须的。

5) 试比较用最大 WBLT、最大 HBLT 及最大堆来实现最大优先队列时各自的优缺点。

13. 在描述一棵最大 HBLT 时，可采用一个指向值为 MinElement 节点的指针（见练习 10）来代替 NULL（或 0）指针。根据这一变化来修改最大 HBLT 的代码。新代码是否比原代码执行得更快？

## 9.5 应用

## 9.5.1 堆排序

你也许已注意到可利用堆来实现  $n$  个元素的排序，所需时间为  $O(n \log n)$ 。可先将要排序的  $n$  个元素初始化为一个最大堆，然后每次从堆中提取（即删除）元素。各元素将按递减次序排列。初始化所需要的时间为  $\Theta(n)$ ，每次删除所需要的时间为  $O(\log n)$ ，因此总时间为  $O(n \log n)$ ，这个时间要比第 2 章中的排序算法（时间为  $O(n^2)$ ）好得多。

上述的排序策略称为堆排序，其实现代码见程序 9-12。



程序9-12 堆排序

```

template <class T>
void HeapSort(T a[], int n)
{// 利用堆排序算法对 a[1:n] 进行排序
    // 创建一个最大堆
    MaxHeap<T> H(1);
    H.Initialize(a,n,n);

    // 从最大堆中逐个抽取元素
    T x;
    for (int i = n-1; i >= 1; i--) {
        H.DeleteMax(x);
        a[i+1] = x;
    }

    // 在堆的析构函数中保存数组 a
    H.Deactivate();
}

```

图9-9给出了程序9-12的for 循环中最初几个*i* 值的运行过程，循环开始时的最大堆如图9-5d所示。

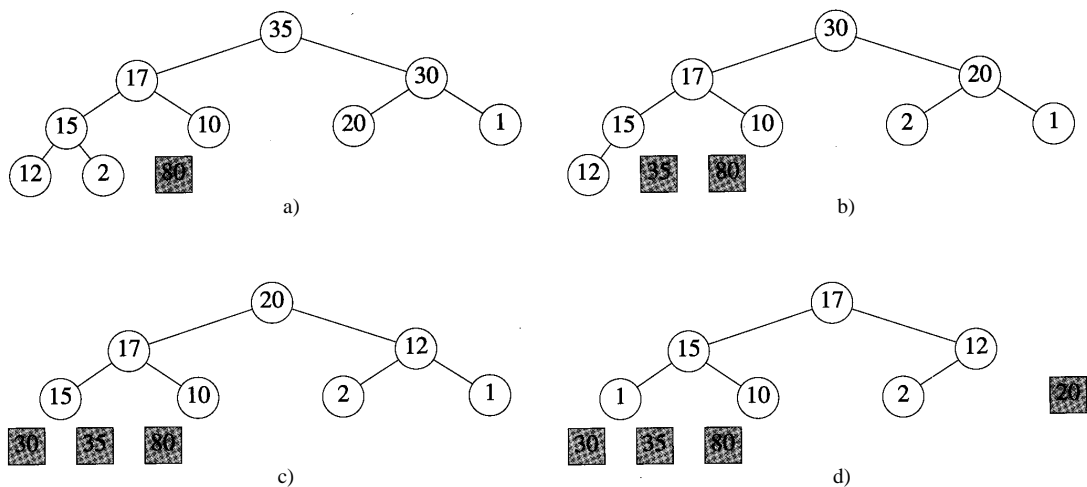


图9-9 堆排序

### 9.5.2 机器调度

考察一个机械厂，其中有  $m$  台一模一样的机器。现有  $n$  个作业需要处理，设作业  $i$  的处理时间为  $t_i$ ，这个时间为从将作业放入机器直到从机器上取下作业的时间。所谓调度 (schedule) 是指按作业在机器上的运行时间对作业进行分配，使得：

- 一台机器在同一时间内只能处理一个作业。
- 一个作业不能同时在两台机器上处理。
- 作业  $i$  一旦运行，则需要  $t_i$  个时间单位。

假设每台机器在0时刻都是可用的,完成时间(或调度长度)是指完成所有作业的时间。在一个非抢先调度(nonpreemptive schedule)中,作业从 $s_i$ 时刻起在某台机器上进行处理,其完成时刻为 $s_i + t_i$ ,这里仅考虑非抢先调度。

图9-10给出了七个作业在三台机器上进行调度的情形,七个作业所需时间分别为(2, 14, 4, 16, 6, 5, 3)。三台机器分别被编号为M1、M2和M3。每个阴影区代表作业的运行区间,阴影区的编号代表作业的索引号。作业4在0到16时刻被调度到机器1(M1)上运行,在这16个时间单位中,机器1完成了对作业4的处理。作业2在0到14时刻被调度到机器2上处理,之后机器2在14到17时刻处理作业7。在机器3上,作业5在0~6时刻内完成,作业6在6~8时刻内完成,作业3在8~11时刻内完成,作业1在11~15时刻内完成,作业7在15~17时刻内完成。注意到每个作业只能在一台机器上 $s_i$ 从时刻到 $s_i + t_i$ 时刻内完成且任何机器在同一时刻仅能处理一个作业,完成所有作业的时刻为17,因此完成时间或调度长度为17。

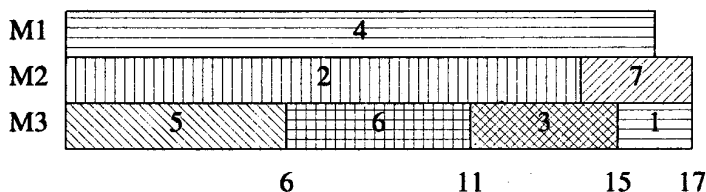


图9-10 三台机器的调度

我们的任务是写一个程序,以便确定如何进行调度才能使在 $m$ 台机器上执行给定的 $n$ 个作业时所需要的处理时间最短。建立这种调度非常难。实际上,没有人能够设计一个具有多项式时间复杂性的算法(即一个复杂性为 $O(n^k m^l)$ 的算法, $k$ 和 $l$ 为常数)来解决最小调度时间问题。

调度问题是著名的NP-复杂问题(NP表示nondeterministic polynomial)中的一种。NP-复杂及NP-完全问题是指尚未找到具有多项式时间复杂性算法的问题。NP-完全问题是一类判断问题,也就是说,这类问题的答案为是或否。机器调度问题不是一个判断问题,因为它的答案是给出作业在机器间的分配方案(使完成时间最少)。可以构造另一类机器调度问题,除了给定任务和机器外,还给定了时间TMin,这类问题要求确定是否存在一种调度仅需TMin或更少的时间,这种问题应属于判断问题,因而是NP-完全问题。NP-复杂问题可以是判断问题,也可以不是判断问题。

成千上万的具有实际意义的问题都是NP-复杂或NP-完全问题,如果有人能对一个NP-复杂或NP-完全问题找到一个多项式算法,那么他/她同时也找到了能在多项时间内解决所有NP-复杂或NP-完全问题的方法。虽然不能证明NP-完全问题不能在多项式时间内得到解决,但大家都认为这已是一个事实。因此,NP-复杂问题的优化问题经常用近似算法(approximation algorithms)解决,虽然近似算法不能保证得到最优解,但能保证所获得的是近似最优解。

在调度问题中,采用了一个称为最长处理时间优先(longest processing time first, LPT)的简单调度策略,它可以帮助我们获得一个较理想的调度长度,该长度为最优调度长度的 $4/3 - 1/(3m)$ 。在LPT算法中,作业按它们所需处理时间的递减顺序排列。在分配一个作业时,总是将其分配给最先变为空闲的机器。

如图9-10的例子,为了获得一个LPT调度,首先根据作业所需处理时间把作业按递减次序排列,所得排列结果为(4,2,5,6,3,7,1)。首先为作业4分配机器,由于三台机器均是从0时刻开

始可用，作业4可分配给任何一台。假设将其分配给机器1，则机器1直到第16时刻才可用。下面可分配作业2，可以将其分配给机器2或机器3，假设将作业2分配给机器2，这样机器2直到第14时刻才可用。然后在0~6时刻将作业5分配给机器3处理。对于下一个待分配的作业6，最先可用的机器为机器3，它在6时刻变为空闲，因此将作业6在6~11时刻分配给机器3。机器3下一次可用的时间为第11时刻，因此可在11时刻调度作业3。按此方法继续下去，即可得到图9-10的调度方案。

定理9-2 [Graham] 令 $F^*(I)$ 为在 $m$ 台机器上执行作业集合 $I$ 的最佳调度完成时间， $F(I)$ 为采用LPT调度策略所得到的调度完成时间，则

$$\frac{F(I)}{F^*(I)} \leq \frac{4}{3} - \frac{1}{3m}$$

证明 参见1996年纽约计算机科学出版社出版的 *Computer Algorithm C++*，作者E.Horowitz, S.Sahni和S.Rajasekeran。

实际上，LPT调度比定理9-2所给界限更接近最佳算法。利用堆可在 $O(n \log n)$ 时间内建立LPT调度方案。首先，当 $n = m$ 时，只需要将作业 $i$ 在 $0 \sim t_i$ 时刻内分配到机器 $i$ 上去处理。当 $n > m$ 时，可以首先利用HeapSort（见程序9-12）将作业按处理时间递增的顺序排列。为了建立LPT调度，作业按相反次序进行分配。为决定作业分配给哪一台机器，必须知道哪台机器最先可用。为此，维持一个 $m$ 台机器的最小堆，这个最小堆的每个元素为MachineNode类型（见程序9-13）。avail是机器可用的时刻，ID为机器编号。DeleteMin用来获取最先可用的机器。当机器的可用时间增加后，需将其再次插入到最小堆中。由于所有机器的最初可用时间都为0时刻，所以这些机器的avail值都为0。程序9-14给出了实现代码。类型T至少必须包括time与ID域。程序9-13为\*a和T提供了一个比较合适的类型JobNode。ID为每个作业的唯一标识，time为每个作业所需的处理时间。

程序9-13 JobNode 及MachineNode 数据类型

---

```

class JobNode {
    friend void LPT(JobNode *, int, int);
    friend void main(void);
public:
    operator int () const {return time;}
private:
    int ID, // 作业号
        time; // 处理时间
};

class MachineNode {
    friend void LPT(JobNode *, int, int);
public:
    operator int () const {return avail;}
private:
    int ID, // 机器号
        avail; // 何时变空闲
};

```

---

程序9-14 构造LPT调度

```
template <class T>
void LPT(T a[], int n, int m)
{// 构造一个有 m 台机器的 LPT 调度
    if (n <= m) {
        cout << "Schedule one job per machine." << endl;
        return;}

    HeapSort(a,n); // 按升序排列
    // 对 m 台机器及最小堆进行初始化
    MinHeap<MachineNode> H(m);
    MachineNode x;
    for (int i = 1; i <= m; i++) {
        x.avail = 0;
        x.ID = i;
        H.Insert(x);
    }

    // 构造调度
    for (i = n; i >= 1; i--) {
        H.DeleteMin(x); // 获取第一台空闲的机器
        cout << "Schedule job " << a[i].ID
            << " on machine " << x.ID << " from "
            << x.avail << " to "
            << (x.avail + a[i].time) << endl;
        x.avail += a[i].time; // 新的可用时间
        H.Insert(x);
    }
}
```

### LPT 复杂性分析

当  $n \leq m$  时，LPT函数所花费时间为  $\Theta(1)$ 。当  $n > m$  时，堆排序花费时间为  $O(n \log n)$ 。虽然在堆的初始化时做了  $m$  次插入，但由于所有元素具有相同的值，所以每次插入实际开销为  $\Theta(1)$ ，因此初始化所花总时间为  $\Theta(m)$ 。在第二个for循环中，执行了  $n$  次DeleteMin和  $n$  次Insert操作，每次需  $O(\log m)$  的时间，因此需时  $O(n \log m)$ 。所以总的时间为： $O(n \log n + n \log m) = O(n \log n)$ （因为  $n > m$ ）。

### 9.5.3 霍夫曼编码

在7.5.2节中介绍了一种基于LZW算法的文本压缩工具，这种算法利用了字符串在文本中重复出现的规律。霍夫曼编码（Huffman code）是另外一种文本压缩算法，它根据不同符号在一段文字中的相对出现频率来进行压缩编码。假设文本是由  $a, u, x, z$  组成的字符串，若这个字符串的长度为1000，每个字符用一个字节来存贮，共需1000个字节（即8000位）的空间。如果每个字符用2位二进制来编码（ $00=a, 01=x, 10=u, 11=z$ ），则用2000位二进制即可表示1000个字符。此外，还需要一定的空间来存放编码表，可采用如下格式来存储：

符号个数，代码1，符号1，代码2，符号2，...

符号个数及每个符号分别用 8 位二进制来表示, 每个代码需占用  $\lceil \log_2 (\text{符号个数}) \rceil$  位二进制。因此, 在本例中, 代码表需占用  $5 \times 8 + 4 \times 2 = 48$  位, 压缩比为  $8000/2048 = 3.9$ 。

利用上述编码方法, 字符串 *aaxuaxz* 的压缩编码为二进制串 00000110000111, 每个字符的编码具有相同的位数 (两位)。从左到右依次从位串中取出 2 位, 通过查编码表便可获得原字符串。

在字符串 *aaxuaxz* 中, *a* 出现了三次。一个字符出现的次数称为频率 (frequency), *a, x, u, z* 在这个字符串中出现的频率分别为 3, 2, 1, 1。当每个字符出现的频率有很大变化时, 可以通过可变长的编码来降低每个位串的长度。如果使用编码 ( $0=a, 10=x, 110=u, 111=z$ ), 则 *aaxuaxz* 的压缩编码为 0010110010111。编码长度为 13 位, 比原来每个字符用 2 位、总长为 14 位要稍好一些。当频率相差大时这种差别会更为明显。如果四个字符的出现频率分别为 (996, 2, 1, 1), 则每个字符用 2 位编码所得到编码的长度为 2000 位, 而用可变长编码则仅为 1000 位。

但是怎样对位串进行解码呢? 若每个代码为 2 位, 则解码很容易——只需每次取出 2 位并利用编码表来得到这两位代表什么。若使用可变长编码, 则并不知道每次应取出多少位, 字符串 *aaxuaxz* 经编码为 0010110010111, 当从左至右解码时, 必须知道第一个字符的代码是 0, 00 还是 001。由于没有哪个字符的代码以 00 打头, 因此第一个字符的代码必为 0, 根据编码表可知该字符为 *a*。下一个代码为 0, 01 或 010, 同理, 由于不存在以 01 打头的字符代码, 因此代码必为 0, 相应元素为 *a*。根据这种方法不断进行下去, 就可以对整个位串进行解码。

为什么能够采用上述方法进行解码呢? 通过仔细观察所使用的 4 种代码 (0, 10, 110, 111), 可以发现没有任何一个代码是另一代码的前缀。因此, 当从左到右检查代码时, 可以很确定地得到与实际代码相匹配的字符。

可以利用扩充二叉树 (见 9.4.1 节定义) 来派生一个实现可变长编码的特殊类, 该类满足上述前缀性质, 被称为霍夫曼编码。

在扩充二叉树中, 可对从根到外部节点的路径进行编码, 方法是向左孩子移动时取 0, 向右孩子移动时取 1。在图 9-6b 中从根到外部节点 *b* 的路径编码为 010, 到节点 (*a, b, c, d, e, f*) 的路径编码分别为 (00, 010, 011, 100, 101, 11)。注意到每条从根到外部节点的路径的编码不会是另一条路径编码的前缀, 因此, 这种编码可用来对字符 *a, b, ..., f* 分别编码。令 *s* 为由这些字符组成的字符串,  $F(x)$  为字符  $x (x \in \{a, s, c, d, e, f\})$  的频率。若利用上述代码对 *s* 进行编码, 则编码后的串长为:

$$2 * F(a) + 3 * F(b) + 3 * F(c) + 3 * F(d) + 3 * F(e) + 2 * F(f)$$

对于一棵具有外部节点 1, ..., *n* 的扩充二叉树, 对应的压缩编码串的长度为:

$$WEP = \sum_{i=1}^n L(i) * F(i)$$

其中  $L(i)$  为从根到达外部节点 *i* 的路径长度 (即路径的边数); WEP 为二叉树的加权外部路径长度 (weighted external path length)。为了减小压缩编码串的长度, 必须利用二叉树中的编码, 其中二叉树的外部节点对应于字符串中被编码的字符, 且它的加权外部路径长度最小。若二叉树对于给定的频率具有最小加权外部路径长度, 则这棵树被称为霍夫曼树 (Huffman tree)。

为了利用霍夫曼编码对字符串或一段文本进行压缩编码, 必须:

- 1) 获得不同字符的频率。
- 2) 建立具有最小加权外部路径的二叉树 (即霍夫曼树), 树的外部节点用字符串中的字符表示, 外部节点的权重 (weight) 即为该字符的频率。
- 3) 遍历从根到外部节点的路径得到每个字符的编码。

4) 用字符的编码来代替字符串中的字符。

为了方便解码,需要保存字符代码映射表或每个字符的频率表。对于后一种情况,可以采用方法2)来重构霍夫曼编码树,以得到相应的霍夫曼编码。后面将详细讨论方法2)。

为了构造霍夫曼树,首先从仅含一个外部节点的二叉树集合开始,每个外部节点代表字符串中一个不同的字符,其权重等于该字符的频率。此后不断地从集合中选择两棵具有最小权重的二叉树,并把它们合并成一棵新的二叉树,合并方法是把这两棵二叉树分别作为左右子树,然后增加一个新的根节点。新二叉树的权重为两棵子树的权重之和。这个过程可一直持续到仅剩下一棵树为止。

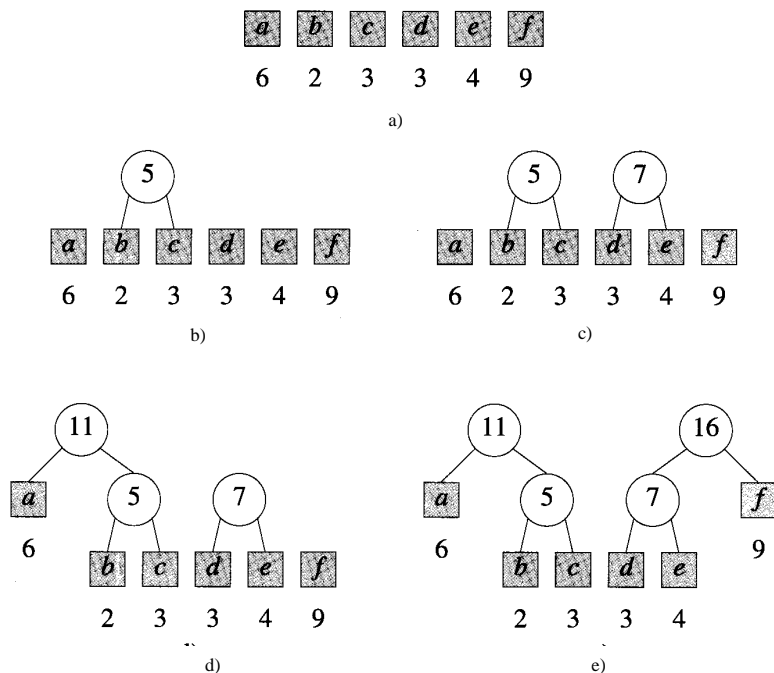


图9-11 建立霍夫曼树

a) 初始集合 b) 第一次合并之后 c) 第二次合并之后 d) 第三次合并之后 e) 第四次合并之后

下面利用上述方法来构造由6个字符( $a, b, c, d, e, f$ )构成的霍夫曼树,这6个字符的频率分别为(6,2,3,3,4,9)。初始的二叉树集合如图9-11a所示,方框外部的数字为树的权重。首先选择具有最小权重的树 $b$ 和具有次小权重的树 $c$ ,将 $b$ 与 $c$ 合并,所得到的结果如图9-11b所示。接下来选择具有最小权重的两棵树 $d$ 和 $e$ 进行合并,合并后的结果如图9-11c所示。从图9-11c的四棵树中选出具有最小权重的树 $a$ 和权重为5的树,将它们合并后得到权重为11的树。从剩下的三棵树(如图9-11d所示)中选取树 $f$ 及权重为7的树,并进行合并。至此,还剩下图9-11e所示的两棵树,将这两棵树合并后即得到图9-6b所示的二叉树,其权重为27。

定理9-3 上述过程所建立的二叉树具有最小加权外部路径。

证明 留作练习(练习19)。

霍夫曼树的建立过程可以利用最小堆来实现,最小堆用来存贮二叉树集合。最小堆中的每个元素包括一棵二叉树及其权重值,二叉树本身是8.8节所定义的BinaryTree类的一个成员。

对于外部节点其 data 域设置为它所代表的字符，内部节点的 data 域设置为 0。为了方便，可以假设字符用数字 1 到 n 表示。程序 9-15 中的 HuffmanTree 函数假定已经定义了类 Huffman（见程序 9-16）。

程序 9-15 建立霍夫曼树

```
template <class T>
BinaryTree<int> HuffmanTree(T a[], int n)
{
    // 根据权重 a[1:n]构造霍夫曼树
    // 创建一个单节点树的数组
    Huffman<T> *w = new Huffman<T> [n+1];
    BinaryTree<int> z, zero;
    for (int i = 1; i <= n; i++) {
        z.MakeTree(i, zero, zero);
        w[i].weight = a[i];
        w[i].tree = z;
    }

    // 把数组变成一个最小堆
    MinHeap<Huffman<T> > H(1);
    H.Initialize(w,n,n);

    // 将堆中的树不断合并
    Huffman<T> x, y;
    for (i = 1; i < n; i++) {
        H.DeleteMin(x);
        H.DeleteMin(y);
        z.MakeTree(0, x.tree, y.tree);
        x.weight += y.weight; x.tree = z;
        H.Insert(x);
    }

    H.DeleteMin(x); // 最后的树
    H.Deactivate();
    delete [] w;
    return x.tree;
}
```

程序 9-16 Huffman 类

```
template<class T>
class Huffman {
    friend BinaryTree<int> HuffmanTree(T [], int);
public:
    operator T () const {return weight;}
private:
    BinaryTree<int> tree;
    T weight;
};
```



HuffmanTree输入 $n$  个频率（即权重，存放在数组 $a$  中）的集合并返回一个霍夫曼树。它首先建立 $n$  棵二叉树，每棵树仅由一个外部节点构成。这些树用数组 $w$  来存贮，后面将把 $w$ 初始化为一个最小堆。第二个for循环从最小堆中取出权重最小的两棵二叉树并将它们合并成一棵二叉树，然后将结果插入最小堆中。

HuffmanTree 函数的复杂性

当 $T$ 为内部数据类型时，构造和删除数组 $w$  所需时间为 $\Theta(1)$ ，而当 $T$ 为用户自定义的类时需耗时 $\Theta(n)$ 。第一个for循环和堆的初始化需 $\Theta(n)$  时间。第二个for循环中，总共执行了 $2(n-1)$ 次删除最小元素及 $n-1$ 次插入操作，需 $O(n\log n)$  的时间。函数其余部分花费的时间为 $\Theta(1)$ 。因此HuffmanTree函数总的时间复杂性为 $O(n\log n)$ 。

## 练习

14. 比较在最坏情况下堆排序与插入排序的执行时间。对于堆排序，利用一些随机序列来估计最坏情况下的执行时间。当 $n$  取什么值时，堆排序比插入排序所用时间少？

15. 利用练习9与10的思想，实现一种比程序9-12更块的堆排序。利用随机数据比较两种实现的执行时间。

16. 一种稳定的排序算法是指具有相同值的记录在排序前与排序后具有相同的顺序。假设记录3与记录10的关键值相同，对于一个稳定排序，排序后记录3仍在记录10的前面。请问堆排序是稳定排序吗？插入排序呢？

17. 在程序9-14中，第二个for循环每次执行一次最小元素删除和一次插入，这两个操作使最小关键值的量得以增加，所增加的量为刚被调度的作业的处理时间。可以利用一个扩充的最小优先队列来加快程序9-14的执行。这种扩充包括最小优先队列通常支持的函数以及函数IncreaseMinkey( $x, e$ )，后者用于将最小关键值增加 $x$  并将结果返回到 $e$  中， $e$  即原来的含有最小关键值的元素。IncreaseMinkey 函数首先将根的值增加 $x$ ，然后沿堆向下移动（尤如最小元素删除操作），并将元素 $e$  上移直到找到一个合适的位置。

1) 设计一个新类 ExtendedMinHeap，提供 MinHeap类中出现的所有成员函数及IncreaseMinkey函数。template<class Te, class Tk> class ExtendMinHeap 应从类MinHeap<Te> 中派生而来。Te 为元素的数据类型，Tk 为关键值的数据类型，因此在IncreaseMinKey( $x, e$ )中， $x$  是Tk类型， $e$  为Te 类型。可以假设操作符 $+=$ 已被重载，因此 $e += x$  语句表示将 $e$  的关键值增加 $x$ 。

2) 利用函数IncreaseMinKey重写程序9-14。

3) 比较代码与程序9-14的代码。

18. 将 $n$  件物品装入容器，第 $i$  件占用的空间为 $s_i$ ，且每个容器的容量为 $c$ 。装入过程采用最不合适法则（worst-fit rule），每次只能给容器分配一件物品。在分配物品时，寻找具有最大剩余容量的容器，若容器可以装下这件物品，则可分配，否则需使用一个新的容器。

1) 编写一个程序，输入为 $n$ ， $s_i$  和 $c$ ，输出为物品在容器中的分配方案。利用最大堆来记录容器的可用空间。

2) 程序的复杂性是多少？（复杂性应为 $n$  及容器个数 $m$  的函数）

\*19. 利用外部节点的数目进行归纳证明定理9-3。在归纳步中可以假定存在一棵二叉树，该树拥有最小加权外部路径，且存在一棵子树，子树中有一个内部节点和两个外部节点，外部节点分别对应两个最小频率。

20. 写一个程序，其输入为HuffmanTree函数（见程序9-15）所建立的霍夫曼树，输出为编码表。程序的复杂性为多少？

\*21. 设计一个完整的，基于霍夫曼编码的压缩-解压缩软件包，并用适当的文本文件来检验其正确性。

\*22. 欲存储0到511之间的 $n$ 个整数，利用霍夫曼编码编写一个压缩-解压缩包来实现。

23. run 是一个有序元素序列。假设两个run 可以在  $\Theta(r+s)$  时间内合并为一个run，其中 $r$ 与 $s$ 分别为要合并的两个run 的长度。通过不断地合并两个run，可将 $n$ 个不同长度的run 最终合并成一个。解释如何运用霍夫曼树来合并 $n$ 个run，以使时间开销最小。

## 9.6 参考及推荐读物

想更详细地了解优先队列及其各种变化，可参考E.HoroWitz, S.Sahni, D.mehta. *Fundamentals of Data Structures in C++*. W.H.Freeman, 1994。

高度优先左高树可参考专题论文 R.Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983。权重优先左高树可参见论文 S.Cho, S.Sahni. Weight Biased Leftist Trees and Modified Skip Lists. *Proceedings, Second International Conference COCOON '96*, Lecture Notes in Computer Science, Springer Verlag 1090, 1996, 361~370。

可以从 *Computer and Intractability* 一书中找到更多的 NP-复杂问题，如：M.Gorey, D.Johnson. *A Guide to the Theory of NP-completeness*. W.H.Freeman, 1979; E.Horowitz, S.Sahni, S.Rajasekeran. *Computer Algorithms/C++*. Computer Science, 1996。

## 第10章 竞赛树

本章将介绍一类新的树，称为竞赛树。象 9.3 节的堆一样，竞赛树也是完全二叉树，它可用 8.4 节定义的公式化描述的二叉树来进行最有效地存储。竞赛树支持的基本操作是替换最大（或最小）元素。如果有  $n$  个元素，操作的开销为  $O(\log n)$ 。虽然利用堆和左高树也能获得同样的复杂性  $O(\log n)$ ，但它们都不能实现可预见的断接操作。所以当需要按指定的方式断开连接时，可选择竞赛树这种数据结构。比如选择最先插入的元素，或选择左端元素（假定所有元素按从左到右的次序排列）。

本章将研究两种竞赛树：赢者树和输者树。尽管赢者树更直观、更接近现实，但输者树的实现更高效。本章的应用部分将考察另一种 NP-复杂问题——箱子装载。我们将利用竞赛树来高效地实现解决箱子装载问题的两个近似算法。试一试能否用本书迄今为止所介绍的其他任意一种数据结构以相同的时间复杂性来实现这两个算法，从中你会得到一些有益的启示。

### 10.1 引言

假定在一次乒乓球比赛中有  $n$  名选手，该比赛采用的是突然死亡（sudden-death）的比赛规则，即一名选手只要输一场球，就被淘汰，最终必然只剩下一个赢者。定义此“幸存”的选手为竞赛赢家。图 10-1 给出由  $a \sim h$  共 8 名选手参加的某次比赛。图中用一棵二叉树来描述整个比赛，每个外部节点分别代表一名选手，每个内部节点分别代表一场比赛，参加每场比赛的选手是子节点所对应的两名选手。这里，同一层节点所构成的一轮比赛可以同时进行。在第一轮比赛中  $a$  与  $b$ 、 $c$  与  $d$ 、 $e$  与  $f$ 、 $g$  与  $h$  交战。每场比赛的赢者记录在代表该场比赛的内部节点中。在图 10-1a 的例子中，第一轮比赛的 4 个赢家为  $b$ 、 $d$ 、 $e$  和  $h$ ，因而  $a$ 、 $c$ 、 $f$ 、 $g$  被淘汰。下一轮比赛在  $b$  与  $d$ 、 $e$  与  $h$  之间进行，胜者为  $b$  和  $e$ ，故由  $b$ 、 $e$  参加最后的决赛，最终赢者为  $e$ 。图 10-1b 给出有  $a \sim e$  共 5 名选手参加的比赛，最后的赢家为  $c$ 。

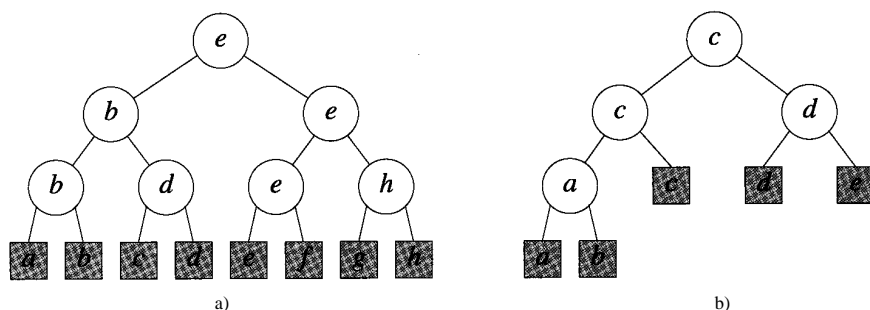


图10-1 竞赛树

a) 8名选手 b) 5名选手

虽然图 10-1 的两棵树都是完全二叉树（实际上树 a 还是满二叉树），但在现实中竞赛问题所对应的树不一定是完全二叉树。然而，使用完全二叉树可以减小比赛的场次。对于  $n$  名选手的比赛，比赛场次为  $\lceil \log_2 n \rceil$  个。因为图 10-1 中竞赛树的每一个内部节点都记录了对应比赛的赢

家，所以称之为赢者树（winner tree）。10.4节将介绍另一类型的树，称为输者树（loser tree），故名思义，它的每一个内部节点记录的是比赛的输者。竞赛树在某些情况下也被称为选择树（selection tree）。

为适应计算机的实现，需要对赢者树作一些适当的修改。不妨假定赢者树为完全二叉树。

**定义 [赢者树]** 对于 $n$ 名选手，赢者树是一棵含 $n$ 个外部节点， $n-1$ 个内部节点的完全二叉树，其中每个内部节点记录了相应赛局的赢家。

为决定一场比赛的赢家，假设每个选手有一得分且赢者取决于对两选手得分的比较。在最小赢者树（min winner tree）中，得分小的选手获胜；同理，在最大赢者树（max winner tree）中，得分大的选手获胜。有时，也可用左孩子对应的选手代表赢家节点。图10-2a 给出含8名选手的最小赢者树，而图10-2b 给出含5名选手的最大赢者树。每个外部节点之下的数字表示选手得分。

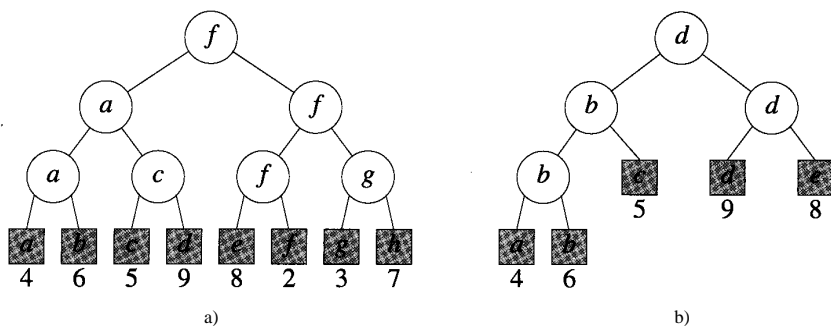


图10-2 赢者树

a) 最小赢者树 b) 最大赢者树

赢者树的一个优点在于即使一名选手得分改变了，也可以较容易地修改此树。如当选手 $d$ 的值由9改为1时，只需沿从 $d$ 到根那条路径修改二叉树，而不必改变其他比赛的结果。有时甚至还可以避免重赛某些场次。如在图10-2a中当 $b$ 值由6改为5时，其父节点的比赛结果中 $b$ 仍为输家，故此时所有比赛的结果未变，因此不必重赛 $b$ 的祖父及曾祖父节点所对应的比赛。

对于一棵有 $n$ 名选手的赢者树，当一个选手的得分发生变化时，需要修改的比赛场次介于 $0 \sim \log_2 n$ 之间，因此重构赢者树需耗时 $O(\log n)$ 。此外，由于 $n$ 名选手的赢者树在内部节点中共需进行 $n-1$ 场比赛（按从最低层到根的次序进行），因此赢者树的初始化时间为 $\Theta(n)$ 。也可以采用前序遍历方式来完成初始化，方法是在每一访问步中安排一场比赛。

**例10-1 [排序]** 可用一个最小赢者树在 $\Theta(n \log n)$ 时间内对 $n$ 个元素进行排序。首先，用 $n$ 个元素代表 $n$ 名选手对赢者树进行初始化。利用元素的值来决定每场比赛的结果，最后的赢家为具有最小值的元素，然后将该选手（元素）的值改为最大值（如 $\infty$ ），使它赢不了其他任何选手。在此基础上，重构该赢者树，所得到的最终赢家为该排序序列中的下一个元素。以此类推，可以完成 $n$ 个元素的排序，时耗为 $\Theta(n)$ 。每次改变竞赛赢家的值并重构赢者树的时耗为 $\Theta(\log n)$ ，这个过程共需执行 $n-1$ 次，因此整个排序过程所需要的时间为 $\Theta(n \log n)$ 。

**例10-2 [run 的产生]** 本书前几章所讨论的排序方法（插入排序、堆排序等）都属内部排序法（internal sorting method），待排序的各个元素必须放入计算机内存中。当待排序元素所需的空

间超出内存的最大容量时，内部排序法就需要与存储了部分或所有元素的外部存储介质（如磁盘）打交道，致使排序效率不高。在这种情况下必须采用外部排序法（external sorting method）。外部排序一般包括两步：1) 产生部分排序结果run；2) 将这些run合并在一起得到最终的run。

假设要为含16 000个元素的记录排序，且在内部排序中一次可排序1000个记录。为此在第1)步中，重复以下步骤16次，可得到16个排序结果(run)：

输入1000个记录

用内部排序法对这1000个记录进行排序

输出排序结果run

接下来需产生最终的排序结果，即完成上述第2)步。在本步中要重复地将 $k$ 个run合并成一个run。如在本例中有16个run（如图10-3所示），它们的编号分别为 $R_1, R_2, \dots, R_{16}$ 。首先，将 $R_1 \sim R_4$ 合并得到 $S_1$ ，其长度为4000个记录，然后将 $R_5 \sim R_8$ 合并，以此类推。接下来将 $S_1 \sim S_4$ 合并得到 $T_1$ ， $T_1$ 便是外部排序的最终结果。

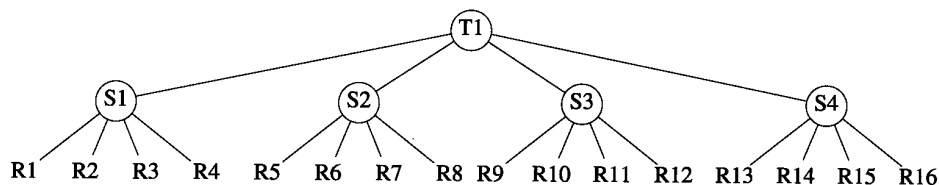


图10-3 16个run的四路合并

一种合并 $k$ 个run的简单方法是：从 $k$ 个run的前面不断地移出值最小的元素，该元素被移至输出run中。当所有的元素从 $k$ 个输入run移至输出run中时，便完成了合并过程。注意到在选择输出run的下一元素时，只需要知道内存中每个输入run的前面元素的值。故只要有足够内存保存 $k$ 个元素值，就可合并任意长度的 $k$ 个run。在实际应用上，我们感兴趣的是能一次输入/出更多元素以减少输入/出的次数。

在上面所列举的16 000个记录的例子中，每个run有1000个记录，而内存容量亦为1000个记录。为合并前四个run，可将内存分为五个缓冲区，每个容量为200个记录。其中前四个为输入run的缓冲区，第五个为输出run的缓冲区，用于存放合并的记录。从四个输入run中各取200个记录放入输入缓冲区中，这些记录被不断地合并并放入输出缓冲区中，直到以下条件发生：

- 1) 输出缓冲区已满。
- 2) 某一输入缓冲区空。

当第一个条件发生时，将输出缓冲区内容写入磁盘，写完之后继续进行合并。当第二个条件发生时，则从对应于空缓冲区的输入run中继续读取记录放入该缓冲区，读取过程结束后合并继续进行。当这些run中的4000个记录都写入输出run中时，四个run的合并过程结束（更详细的描述参见E.Horowitz, S.Sahni, D.Mehta. *Fundamentals of Data Structure in C++*. Computer Science Press, 1995。）

run合并所需时间的决定因素之一是第1)步所产生的run的个数。通过使用赢者树，可减少run的个数。开始时，用一个含 $p$ 名选手的赢者树，其中每个选手对应输入集合中的一个元素。每个选手有一个值（即对应的元素值）和一个run号。前 $p$ 个元素的run号均为1。当两选手进行比赛时，具有较小run号的选手获胜。在run号相同的情况下，按选手的值进行比较，具有

较小值的元素成为赢家。为产生 run，重复地将最终赢者  $W$  移入它的 run 号所对应的 run 中，再用下一个输入元素  $N$  取代  $W$  原来的位置。如果  $N$  的值大于等于  $W$  的值，则将元素  $N$  作为相同 run 的成员输出（它的 run 号与  $W$  的相同）。如果  $N$  的值小于  $W$  的值，若将  $N$  放入与  $W$  相同的 run 中将会破坏 run 中元素的排序，因此将  $N$  的 run 号设置为  $W$  的 run 号加 1。

当采用上述方法生成 run 时，run 的平均长度约为  $2p$ 。当  $2p$  大于内存容量时，我们希望能得到更少的 run（与上述方法相比）。事实上，倘若输入集合已排好序（或几乎排好序），只需产生最后的 run，则可直接跳到 run 的合并步，即第 2）步。

**例 10-3 [k 路合并]** 在  $k$  路合并（见例 10-2）中， $k$  个 run 要合并成一个排好序的 run。按例 10-2 中所述的方法，进行  $k$  路合并时，因在每一次循环中都需要找到最小值，故将每一个元素合并到输出 run 中需  $O(k)$  时间，因此产生大小为  $n$  的 run 所需要的时间为  $O(kn)$ 。若使用赢者树，则可将这个时间缩短为  $\Theta(k+n\log k)$ 。首先用  $\Theta(k)$  的时间初始化含  $k$  个选手的赢者树。这  $k$  个选手都是  $k$  个被合并 run 中的头一个元素。然后将赢者移入输出 run 中并用相应的输入 run 中的下一个元素替代之。如果在该输入 run 中无下一元素，则需用一个 key 值很大（不妨为  $\infty$ ）的元素替代之。 $k$  次移入和替代赢家共需耗时  $\Theta(\log k)$ 。因此采用赢者树进行  $k$  路合并的总时间为  $\Theta(k+n\log k)$ 。

## 练习

1. 1) 描述如何用最小堆来替代最小赢者树产生 run（见例 10-2），产生 run 中每一元素的时间为多少？
- 2) 在此应用中，最小赢者树与堆相比，有哪些优点和缺点？
2. 1) 在进行  $k$  路合并时（见例 10-3），如何用最小堆替代最小赢者树？
- 2) 在此应用中，最小赢者树与堆相比有哪些优点和缺点？

## 10.2 抽象数据类型 WinnerTree

在定义抽象数据类型 *WinnerTree* 时，假设选手数是固定的，也就是说，初始化选手数为  $n$  的树之后，不再增减选手数。选手本身不是赢者树的组成部分，故组成赢者树的成分是图 10-1 所示的内部节点。因此，赢者树需要支持的操作有：创建空的赢者树、用  $n$  名选手初始化赢者树、返回赢者、在从选手  $i$  到根的路径上重新进行比赛。各操作的定义在 ADT10-1 中给出。

ADT10-1 赢者树的抽象数据类型描述

抽象数据类型 *WinnerTree* {

实例

完全二叉树，每个节点代表相应比赛的赢者，外部节点代表选手

操作

*Create()*：创建一个空的赢者树

*Initialize(a, n)*：对有  $n$  个选手  $a[1:n]$  的赢者树进行初始化

*Winner()*：返回比赛的赢者

*Replay(i)*：选手  $i$  变化时，重组赢者树

}



## 10.3 类WinnerTree

### 10.3.1 定义

假设用完全二叉树的公式化描述方法来定义赢者树。 $n$ 名选手的赢者树需 $n-1$ 个内部节点 $t[1:n-1]$ 。选手（或外部节点）用数组 $e[1:n]$ 表示。故 $t[i]$ 为数组 $e$ 的一个索引而且类型为 $\text{int}$ 。 $t[i]$ 给出了赢者树中节点 $i$ 对应比赛的赢者。图10-4给出了5选手赢者树中各节点与数组 $t$ 和 $e$ 之间的对应关系。

为实现ADT操作，必须能够确定外部节点 $e[i]$ 的父节点 $t[p]$ 。当外部节点的数目为 $n$ 时，内部节点数为 $n-1$ 。最底层最左端的内部节点的编号为 $2^s$ ，这里 $s = \log_2(n-1)$ 。因此最底层的内部节点数为 $n-2^s$ ，最底层的外部节点数 $LowExt$ 为内部节点数的2倍。例如，在图10-4的树中， $n=5$ ， $s=2$ ，最底层最左端的内部节点为 $t[2^s]=t[4]$ ，该层的内部节点数共为 $n-4=1$ 个，最底层外部节点数为2，倒数第2层的最左端的外部节点号为 $LowExt+1$ 。设 $offset=2^{s+1}-1$ ，可知对于任何外部节点 $e[i]$ ，其父节点 $t[p]$ 满足以下公式：

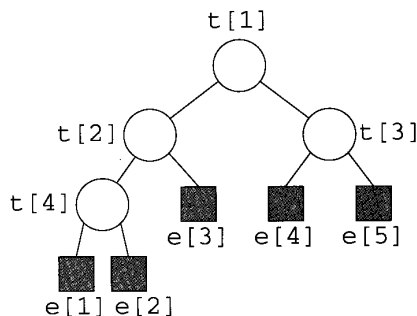


图10-4 树与数组的对应关系

$$p = \begin{cases} (i + offset)/2 & i \leq LowExt \\ (i - LowExt + n - 1)/2 & i > LowExt \end{cases} \quad (10-1)$$

### 10.3.2 类定义

在程序10-1中给出了赢者树的类定义。私有成员包括：MaxSize（允许的最大选手数）、 $n$ （赢者树初始化时的选手数）、 $t$ （内部节点数组）、 $e$ （外部节点数组）、 $LowExt$ 和 $Offset$ 。可以用确定比赛赢者的函数作参数来调用Initialize和Replay。Winner(a,b,c)返回的是 $a[b]$ 和 $a[c]$ 之间比赛的赢者。通过适当定义Winner，可构造最小赢者树、最大赢者树等。

程序10-1 赢者树的类定义

```
template<class T>
class WinnerTree {
public:
    WinnerTree(int TreeSize = 10);
    ~WinnerTree() {delete [] t;}
    void Initialize(T a[], int size, int(*winner)(T a[], int b, int c));
    int Winner() const {return (n) ? t[1] : 0;}
    int Winner(int i) const {return (i < n) ? t[i] : 0;}
    void RePlay(int i, int(*winner) (T a[], int b, int c));
private:
    int MaxSize;
    int n;    //当前大小
```



```

int LowExt; //最底层的外部节点
int offset; //2^k-1
int *t;     //赢者树数组
T *e;      //元素数组
void Play(int p, int lc, int rc, int(*winner)(T a[], int b, int c));
};

```

### 10.3.3 构造函数、析构函数及 Winner函数

在程序 10-2 中，构造函数创建了一个初始为空的赢者树（ $n=0$ ），它可以最多处理  $\text{MaxSize}$  个选手。可用的内部节点为  $t[1] \sim t[\text{MaxSize}-1]$ 。析构函数和  $\text{Winner}$  函数被定义为内部函数，见程序 10-1。

程序 10-2 创建赢者树

```

template<class T>
WinnerTree<T>::WinnerTree(int TreeSize)
{//构造赢者树
    MaxSize = TreeSize;
    t = new int[MaxSize];
    n = 0;
}

```

### 10.3.4 初始化赢者树

程序 10-3 给出了初始化操作的代码。 $a$  为选手数组， $\text{size}$  表示选手数， $\text{winner}$  用于得到  $a[b]$  和  $a[c]$  之间比赛的赢家。在程序中，首先验证赢者树能否处理  $\text{size}$  个选手，如果能，则初始化  $n$  和  $e$ 。接下来计算  $s = \lceil \log_2(n-1) \rceil$ ，再由  $s$  得到  $\text{LowExt}$  和  $\text{offset}$ 。为了完成初始化，从赢者树的外部节点  $i$  开始逐层向上进行比赛，其中  $i$  依次取为  $1, 2, \dots, n$ 。注意在这个过程中，仅当从一个节点的右孩子上升到该节点时，才在该节点进行一场比赛。倘若是从左孩子上升到该节点，因其右子树的赢者尚未确定，因而不能在该节点上进行比赛。在第二个  $\text{for}$  循环中，各比赛由最底层选手（外部节点）来激活。作为右孩子的选手位于  $e[2], e[4], \dots, e[\text{LowExt}]$  中。比赛由保护成员函数  $\text{Play}$ （见程序 10-4）来完成。对于某个  $e[i]$ ，其父节点为  $t[(\text{offset}+i)/2]$ （见公式（10-1）），对手为  $e[i-1]$ 。为了执行由其他  $n-\text{LowExt}$  个选手激活的比赛，必须确定  $n$  是否为奇数。若  $n$  为奇数，则  $e[\text{LowExt}+1]$  为右孩子，否则为左孩子。当  $n$  为奇数时，对手为  $e[t[n-1]]$ ，父节点为  $t[(n-1)/2]$ 。最后的  $\text{for}$  循环语句激活其余外部节点的比赛。

程序 10-3 初始化赢者树

```

template<class T>
void WinnerTree<T>::Initialize(T a[], int size, int(*winner)(T a[], int b, int c))
{//对于数组 a 初始化赢者树
    if (size > MaxSize || size < 2)
        throw BadInput();
    n = size;
}

```

```

e = a;

//计算  $s = 2^{\log(n-1)}$ 
int i, s;
for (s = 1; 2*s <= n-1; s += s);

LowExt = 2*(n-s);
offset = 2*s-1;

//最底层外部节点的比赛
for (i = 2; i <= LowExt; i += 2)
    Play((offset+i)/2, i-1, i, winner);
//处理其余外部节点
if (n % 2) { //当n奇数时，内部节点和外部节点的比赛
    play(n/2, t[n-1], LowExt+1, winner);
    i = LowExt+3;
}
else i = LowExt+2;

//i为最左剩余节点
for (; i <= n; i += 2)
    play((i-LowExt+n-1)/2, i-1, i, winner);
}

```

程序10-4 通过比赛对树进行初始化

```

template<class T>
void WinnerTree<T>::Play(int p, int lc, int rc, int(*winner)(T a[], int b, int c))
{ //在t[p]处开始比赛
    //lc和rc是t[p]的孩子
    t[p] = winner(e, lc, rc);

    //若在右孩子处，则可能有多场比赛
    while (p > 1 && p % 2) { //在右孩子处
        t[p/2] = winner(e, t[p-1], t[p]);
        p /= 2; //到父节点
    }
}

```

函数Play首先在内部节点t[p]处进行比赛，然后沿赢者树不断向上移动并进行比赛，直到从一个左孩子移到其父节点为止。

为了弄清Initialize是如何工作的，现在来考察图10-4所示的5选手例子。在第二个for循环中，e[1]和e[2]之间有一场比赛，该比赛的赢者记录在t[4]中。此时t[2]对应的比赛尚未进行。这是因为t[4]是其父节点t[2]的左孩子。此时n为奇数，因而在t[2]处进行e[t[4]]和e[3]间的比赛。赢者记录在t[2]。在第三个for循环中，首先进行t[3]处的比赛（e[4]和e[5]间的比赛）。因t[3]为其父节点的右子女，故此时t[1]对应的比赛也可进行。

现在来分析函数Initialize的复杂性。s的计算需 $\Theta(\log n)$ 时间。第二次和第三次for循环

(包括函数 Play) 共需  $\Theta(n)$  时间。调用所有 Play(共  $n-1$  次)所需时间为  $\Theta(n)$ , 因此 Initialize 总的复杂性为  $\Theta(n)$ 。

### 10.3.5 重新组织比赛

当选手  $i$  相应的值改变后, 需要重新进行某些甚至所有的从外部节点  $e[i]$  到根  $t[1]$  路径上的比赛。为简单起见, 将再次执行该路径上的所有比赛。实际上, 在例 10-1、10-2 和 10-3 中, 只有赢者的值会发生变化。一个赢者的值发生变化必然会导致重新执行从赢者对应的外部节点开始到根的路径上的所有比赛。程序 10-5 给出了相应的代码。

程序 10-5 当元素  $i$  改变时重新组织比赛

---

```
template<class T>
void WinnerTree<T>::RePlay(int i, int(*winner)(T a[], int b, int c))
{//针对元素i重新组织比赛
    if (i <= 0 || i > n) throw OutOfBounds();

    int p, //比赛节点
        lc, //p的左孩子
        rc; //p的右孩子
    //找到第一个比赛节点及其子女
    if (i <= LowExt) { //从最底层开始
        p = (offset + i)/2;
        lc = 2*p - offset; //p的左孩子
        rc = lc+1; }
    else { p = (i-LowExt+n-1)/2;
        if (2*p == n-1) { lc = t[2*p];
            rc = i; }
        else { lc = 2*p - n + 1 + LowExt;
            rc = lc+1; }
    }
    t[p] = winner(e, lc, rc);

    //剩余节点的比赛
    p /= 2; //移到父节点处
    for (; p >= 1; p /= 2)
        t[p] = winner(e, t[2*p], t[2*p+1]);
}
```

---

为了重新进行比赛, 需要利用公式 (10-1) 来确定第一次比赛的选手。  $t[p]$  处的比赛是  $e[lc]$  和  $e[rc]$  之间的比赛, 赢者记录在  $t[p]$  中。其余比赛都在 for 循环中完成。比赛的总次数为  $\Theta$  (赢者树的高度) =  $\Theta(\log n)$ 。

### 练习

3. 修改函数 RePlay (见程序 10-5), 要求避免进行一些不必要的比赛。通常来说, 当一次比赛的赢者与上一次该比赛的赢者相同时, 可不再进行该场比赛。

4. 编写一个排序程序，该程序用赢者树来重复地将元素插入到已排好序的序列中去（见例10-1）。

## 10.4 输者树

仔细观察赢者树的RePlay操作。不难发现，在许多应用中，只是在前一赢者被新的选手取代后才执行此操作（见例10-1、10-2和10-3）。在这些应用中，从取代赢者的外部节点到根节点的路径上的所有比赛都要重新进行。考察图10-2a中的最小赢者树。当赢者 $f$ 被值为5的选手 $f'$ 取代时，第一场比赛在 $e$ 和 $f'$ 之间进行，其中 $e$ 是该节点处上一次与 $f$ 比赛的输方。而 $f'$ 作为新的赢者，在内部节点 $t[3]$ 上与 $g$ 进行下一场比赛。注意 $g$ 是 $t[3]$ 处与 $f$ 的前一场比赛的输方，而 $t[3]$ 上 $g$ 与 $f'$ 的比赛的赢者为 $g$ 。接下来， $g$ 与根节点上的 $a$ 比赛，而 $a$ 是在根节点处上一场比赛的输方，

如果在每个内部节点中记录的是该节点比赛的输者而不是赢者，那么当 $e[i]$ 发生变化时，就可以减少确定比赛选手的工作量（从节点 $e[i]$ 到根节点的路径上）。最终的赢者可记录在 $t[0]$ 中。图10-5a为对应于图10-2a 8名选手的输者树。现在当赢者 $f$ 的值变成5时，首先移动到它的父节点 $t[6]$ ，对应的比赛为 $e[t[6]]$ 和 $e[6]$ 间的比赛。欲确定 $f'=e[6]$ 的对手，只需检查 $t[6]$ ，而在赢者树中还必须检查 $t[6]$ 的其他子女。在 $t[6]$ 的比赛完成后，输者 $e$ 被记录于此节点中， $f'$ 继续与 $t[3]$ 中前一场比赛的输家 $g$ 进行比赛。此次的输家为 $f'$ ，记录于 $t[3]$ ，赢家 $g$ 则与 $t[1]$ 中上一场比赛的输家 $a$ 比赛，输者为 $a$ ，记录于 $t[1]$ 。新的输者树如图10-5b所示。

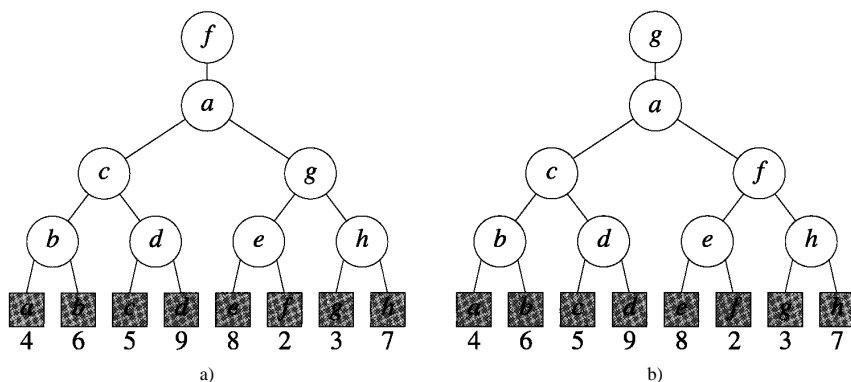


图10-5 8个选手的最小输者树

a) 初始状态 b) 当 $e[b]$ 改变之后

虽然使用输者树可简化一个赢者发生变化后重新比赛的过程，但它并不能简化其他选手发生改变时的情况。例如，当选手 $d$ 的值由9变为3， $t[5]$ 、 $t[2]$ 、 $t[1]$ 上的比赛将重新进行。在 $t[5]$ 处， $d$ 必须与 $c$ 比赛，但 $c$ 不是该节点上一场比赛的输家。在 $t[2]$ 处， $d$ 必须与 $a$ 比赛，但 $a$ 也不是该节点上一场比赛的输家。在 $t[1]$ 处， $d$ 必须与 $f$ 比赛， $f$ 同样不是该节点上一场比赛的输家。为了更容易地重新进行这些比赛，还得用到赢者树。因此仅当 $e[i]$ 为前次比赛的赢家时，对于函数RePlay(i)，采用输者树比采用赢者树执行效率更高。

## 练习

5. 1) 模仿赢者树的类定义（见程序10-1）设计一个C++类LoserTree。比赛赢者可记录在 $t[0]$

中。可定义函数  $\text{RePlay}()$  代替原来的共享成员函数  $\text{RePlay}(i)$ ，该函数从上一次比赛的赢者开始重新组织比赛。

2) 一种对输者树进行初始化的简单方法是先构造一棵赢者树，然后按层次遍历赢者树，遍历过程中用输者替代每个内部节点。遍历从顶层到底层进行。 $t[i]$ 的孩子会告诉我们在 $t[i]$ 处参加比赛的选手，然后根据选手信息确定谁是输家。采用上述策略编写一个初始化函数  $\text{Initialize}$ 。证明代码能在  $\Theta(n)$  的时间内对  $n$  个选手的输者树进行初始化。

3) 使用程序 10-3 中的策略编写  $\text{Initialize}$  函数。尽可能进行比赛并记录比赛输者。当某一比赛无法进行时，记录下该比赛对应的唯一选手。证明代码能在  $\Theta(n)$  时间对含  $n$  个选手的输者树进行初始化。

6. 编写一个排序程序，利用输者树不断地将元素插入到已排好序的序列中。指出程序的复杂性。

## 10.5 应用

### 10.5.1 用最先匹配法求解箱子装载问题

在箱子装载问题中，有若干个容量为  $c$  的箱子和  $n$  个待装载入箱子中的物品。物品  $i$  需占  $s[i]$  个单元 ( $0 < s[i] \leq c$ )。所谓成功装载 (feasible packing)，是指能把所有物品都装入箱子而不溢出，而最优装载 (optimal packing) 则是指使用了最少箱子的成功装载。

例10-4 [卡车装载] 某一运输公司需把包裹装入卡车中，每个包裹都有一定的重量，且每辆卡车也有其载重限制（假设每辆卡车的载重都一样）。在卡车装载问题中，希望用最少的卡车来装载包裹。可将此问题转化为箱子装载问题，即卡车对应于箱子，包裹对应于物品。

例10-5 [集成片分布] 在给定宽度的电路板上按行布设一些电路集成片。集成片高度一致但宽度各不相同。电路板的最小高度由所使用的最小行数决定。集成片分布问题也可转化为箱子装载问题，即电路板的每行对应为一箱子，每个集成片对应为一个需装载的物品。电路板的宽度为箱子容量，而集成片的长度便相当于相应物品的容量。

箱子装载问题和机器调度问题（见 8.5.2 节）一样，也是 NP-复杂问题。因此可用近似的算法求解。在箱子装载问题中，该算法可得到一个接近于最少箱子个数的解。对于箱子装载问题，有 4 种流行的求解算法：

#### 1) 最先匹配法 (First Fit, FF)

物品按  $1, 2, \dots, n$  的顺序装入箱子。假设箱子从左至右排列。每一物品  $i$  放入可盛载它的最左箱子。

#### 2) 最优匹配法 (Best Fit, BF)

设  $c_{\text{Avail}}[j]$  为箱子  $j$  的可用容量。初始时，所有箱子的可负载容量为  $c$ 。物品  $i$  放入具有最小  $c_{\text{Avail}}$  且容量大于  $s[i]$  的箱子中。

#### 3) 最先匹配递减法 (First Fit Decreasing, FFD)

此方法与 FF 类似，区别在于各物品首先按容量递减的次序排列，即对于  $1 \leq i < n$ ，有  $s[i] \geq s[i+1]$ 。

#### 4) 最优匹配递减法 (Best Fit Decreasing, BFD)

此法与 BF 相似，区别在于各物品首先按容量递减的次序排列，即对于  $1 \leq i < n$ ，有  $s[i] \geq s[i+1]$ 。

$s[i+1]$ 。

可以看出, 以上4种方法中没有一种能保证获得最优装载。但这4种方法都很直观实用。

设 $I$ 为箱子装载问题的任一实例。 $b(I)$ 为用于最优装载的箱子数。FF和BF中所用的箱子数不会超过 $\lceil (17/10)b(I) \rceil$ , 而FFD和BFD中的箱子数不会超过 $(11/9)b(I)+4$ 个。以上结论的证明很复杂, 可参考 M.Garey, K.Graham, D.Johnson, A.Yao. Resource Constrained Scheduling as Generalized Bin-Packing. *Journal of Combinatorial Theory, Series A*, 1976, 257~298和D. Johnson, A.Demers, J.Ullman, M.Garey, R.Graham. Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms. *SIAM Journal on Computing*, 1974, 299~325。

**例10-6** 把四件物品 $s[1:4]=[3,5,2,4]$ 放入容量为7的箱子中。使用FF法时, 物品1放入箱子1; 物品2放入箱子2; 此时箱子1还可以放得下物品3, 故物品3放入箱子1; 物品4无法再放入前面这两个箱子, 因此需再使用一个箱子。最后的方案为: 使用三个箱子, 物品1和3放入箱子1; 物品2放入箱子2; 物品4放入箱子3。

若使用BF法, 物品1和2分别放入箱子1和箱子2, 由于箱子2比箱子1更适合放置物品3, 因此将物品3放入箱子2, 这样物品4又可放入箱子1。这种装载方案只用了两个箱子: 物品1、4放入箱子1; 物品2、3放入箱子2。

对于FFD和BFD方法, 首先将物品按2、4、1、3排序, 最后所得装载方案均为: 使用两个箱子, 物品2、3放入箱子1; 物品1、4放入箱子2。

可使用赢者树来实现FF和FFD算法, 所需时间为 $\Theta(n \log n)$ 。因最多会用到 $n$ 个箱子, 故可用 $n$ 个空箱子作为初始条件。设 $avail[j]$ 为箱子 $j$ 的可用空间。初始化时, 对于所有箱子有 $avail[j]=c$ 。接下来, 用 $avail[j]$ 作为选手对最大赢者树进行初始化。图10-6a给出了在 $n=8$ 和 $c=10$ 条件下的最大赢者树。外部节点从左到右对应为箱子1到箱子8。在外部节点之下的数字为该箱子的装载容量。假设 $s[1]=8$ , 为找到装载物品1的最左箱子, 从根 $root[1]$ 开始搜索。根据定义可知 $avail[t[1]] \geq s[1]$ , 因此至少有一个箱子可装载此物品。为找到最左箱子, 要确定在箱子1~箱子4中是否有箱子含足够的空间来装载物品1。当且仅当 $avail[t[2]] \geq s[1]$ 时这些箱子之一有足够空间。在本例中此条件满足, 因此可从根为2的子树开始继续搜索。现在要确定2的左子树(即根为4的子树)所包含的箱子中是否有容量合适的箱子, 若有, 则不必再考虑2的右子树。在本例中, 因 $avail[t[4]] \geq s[1]$ , 故移动到左子树。由于树4的左子树是一个外部节点, 所以可将 $s[1]$ 放入节点4的任一个孩子之中, 若左孩子有足够的空间则将其放入左孩子。当物品1放入箱子1时,  $avail[1]$ 减为2, 然后从 $avail[2]$ 开始重新比赛。新的赢者树如图10-6b所示。现假设 $s[2]=6$ 。因 $avail[t[2]] \geq 6$ , 可知在左子树中有一个箱子有足够的空间, 因此可以先移动到该箱子处, 然后再移到左子树4, 并将物品2放入箱子2。新的结果如图10-6c所示。当 $s[3]=5$ 时, 搜寻将进入根2的子树。对于其左子树,  $avail[t[4]] < s[3]$ , 故根为4的子树所包含的箱子均没有足够空间。因此, 移到右子树5, 并将物品放入箱3。图10-6d给出了相应的结果。接下来, 设 $s[4]=3$ , 从根为4的子树开始搜寻, 因 $avail[t[4]] \geq s[3]$ 故将物品3加入箱子2。

根据前面的讨论, 可以编写采用最先匹配策略的程序。主程序见程序10-6。该程序首先要输入物品数量 $n$ 及每个箱子的容量 $c$ 。假定容量及物品的空间需求均为整数, 并且假定程序只在 $n \geq 2$ 的情况下运行, 因为 $n=0$ 或1时装载方法显而易见。接下来程序要求输入 $n$ 个物品并限制每个物品的空间 $\leq c$ 。最后, 再调用函数FirstFitPack(见程序10-7), 将物品分派到各个箱子中。对于使用FFD策略的程序, 只需将源程序稍作修改, 即在调用FirstFitPack之前按递减顺序对物品进行排序。

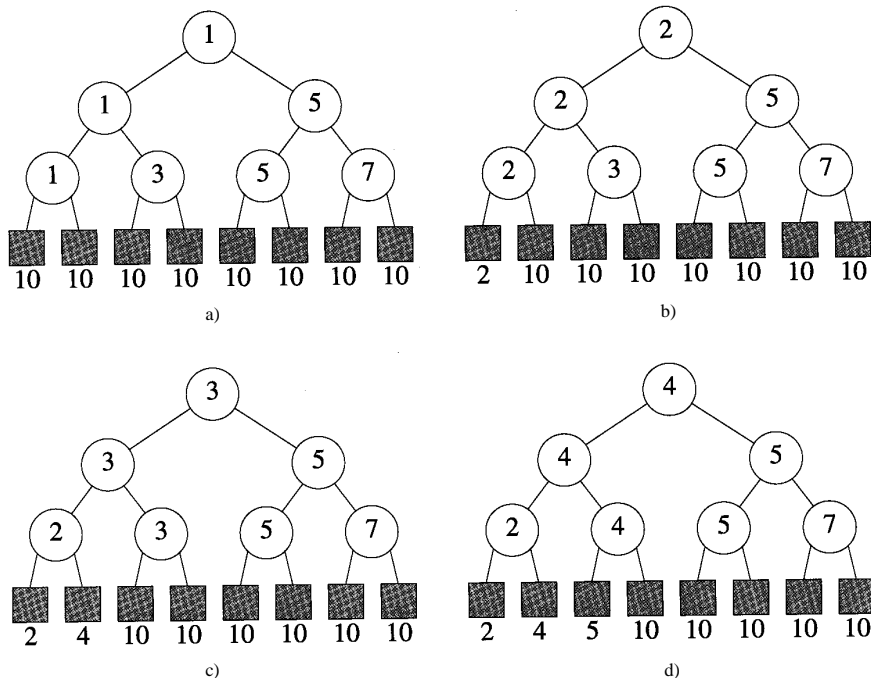


图10-6 最先匹配最大赢者树

a) 初始状态 b) 装载S[1]之后 c) 装载S[2]之后 d) 装载S[3]之后

程序10-6 最先匹配箱子装载法

```

void main(void)
{
    int n, c; // 物品数量和箱子容量
    cout << "Enter number of objects and bin capacity" << endl;
    cin >> n >> c;
    if (n < 2) {cout << "Too few objects" << endl;
        exit(1);}
    int *s = new int[n+1];

    for (int i = 1; i <= n; i++) {
        cout << "Enter space requirement of object" << i << endl;
        cin >> s[i];
        if (s[i] > c) {
            cout << "Object too large to fit in a bin" << endl;
            exit(1);
        }
    }
    FirstFitPack(s, n, c);
}

```

函数FirstFitPack首先对n名选手的最大赢者树进行初始化。选手i代表箱子i当前的容量。所有箱子的容量初始化为c。该函数假定，当比赛开始时左边选手是赢者，除非右边选手比较大。同时还假定WinnerTree（见程序10-1）的类定义中已增加一个共享成员函数：



```
int Winner(int i) const {return(i < n) ? t[i] : 0;}
```

该共享函数用于返回在内部节点  $i$  比赛的赢者。在第二个 for 循环中，物品被依次分派到各箱子中。物品  $i$  的分派过程，是按从根节点到满足该物品的最左箱子的路径进行的。从当前的位置可以判断左子树（根为  $p$ ）是否包含有足够容量的箱子。若无，则确保右子树（根为  $p+1$ ）包含这样的箱子。当然，将优先使用左子树中满足条件的箱子。一旦确定了要移到哪个子树，就把  $p$  修改为其左子树的根，若当前节点的左子树为一外部节点时（即  $p = n$ ），while 循环将结束。注意到我们的程序代码不能准确地记录当前位置。不过，在退出 while 循环时常用  $p$  除以 2 来计算当前位置。当  $n$  为奇数时，当前位置可以是一个外部节点，这时  $p$  等于  $n$ 。对于其他所有情况， $p$  均为内部节点。当  $p$  为外部节点时，该节点对应的箱子是其父节点比赛的赢者，也就是说它是箱子  $t[p/2]$ 。当  $p$  为内部节点时，可以确信  $t[p]$  有足够容量。然而，倘若该箱子不是其父节点的左孩子，它可能不是最左箱子，故我们从该箱子的左边进行检查。一旦确定了用箱子  $b$  来装载物品  $i$ ，该箱子的可用容量应减少  $s[i]$ ，并且沿着赢者树中从该箱子到根的路径重新进行比赛。

程序 10-7 第二个 for 循环中的每次循环需  $\Theta(\log n)$  的时间，因此，该循环共需耗时  $\Theta(n \log n)$ 。该函数其余部分所需的时间为  $\Theta(n)$ ，所以总时耗为  $\Theta(n \log n)$ 。

程序 10-7 函数 FirstFitPack

```
void FirstFitPack(int s[], int n, int c)
{//c为箱子容量
//n为物品数量，s[]为各物品所需要的空间

    WinnerTree<int> *W = new WinnerTree<int>(n);
    int *avail = new int[n+1]; //箱子

    //初始化n个箱子和赢者树
    for (int i = 1; i <= n; i++)
        avail[i] = c; //初始可用的容量
    W->Initialize( avail, n, winner);

    //将物品放入箱子中
    for (int i = 1; i <= n; i++) { //将s[i]放入箱子
        //找到有足够容量的第一个箱子
        int p = 2; //从根的左子树开始查询
        while (p < n) {
            int winp = W->Winner(p);
            if (avail[winp] < s[i]) //第一个箱子在右子树中
                p++;
            p *= 2; //移到左孩子
        }

        int b;
        p /= 2;
        if (p < n) { //在一树节点处
            b = W->Winner(p);
```

```
//若b是右孩子，需要检查箱子b-1。  
//即使b是左孩子，这种检查也没有什么副作用  
if (b > 1 && avail[b-1] >= s[i])  
    b --;  
else //当n为奇数时  
    b = W->Winner (p/2) ;  
  
cout << "Pack object " << i << " in bin " << b << endl;  
avail[b] -= s[i]; //更新可用容量  
w->RePlay(b, winner);  
}  
}
```

### 评价

函数FirstFitPack利用了赢者树建立过程中所规定的某些特有的细节。如，赢者树为用数组表示的完全二叉树，因此能够按数组下标乘2或加1的方式在竞赛树中向下移动。按这种方式在竞赛树中向下移动破坏了使用类的一个目标——信息隐藏。我们希望类的实现细节应对用户透明。当用户与细节隔离时，我们可在保持类的共享成员不发生改变的情况下修改类的具体实现，而这种修改并不会影响应用的正确性。利用信息隐藏的特点，可以扩充 WinnerTree 的类定义，增加从一个内部节点移动至其左、右孩子的共享函数，然后在函数 FirstFitPack 中应用这些函数。

### 10.5.2 用相邻匹配法求解箱子装载问题

在相邻匹配 (Next Fit) 策略中，一次只将一个物品放入一个箱子中。开始时将物品 1 放入箱子 1。对于其余物品，则从最后使用的箱子的下一个箱子开始，用循环的方式轮流查询能够装载该物品的非空箱子。比如，在该轮询法中，若箱子 1 ~ 箱子 b 正在使用，则可认为这些箱子排列成环状。i = b 时，i 的下一个箱子为 i + 1；i = 1 时，i 的下一个箱子为箱子 1。若上一个物品放入了箱子 j，则从箱子 j 的下一个箱子开始不断地查找后续箱子，直到找到具有足够空间的箱子或者又回到箱子 j。若没有找到合适的箱子，则启用一新箱子，并将物品放入该箱子中。

例10-7 欲将6个物品  $s[1:6] = [3, 5, 3, 4, 2, 1]$  放入容量为7的箱子中。用相邻匹配装载法，首先将物品 1 放入箱子 1。物品 2 无合适的箱子，故插入一个新的箱子——箱子 2。对于物品 3，从下一箱子开始搜寻非空的合适箱子。上一次使用的箱子为箱子 2，故下一个箱子为箱子 1。箱子 1 有足够的空间，所以将物品 3 放入箱子 1。对于物品 4，因箱子 1 是上一次使用的箱子，所以从箱子 2 开始轮询。箱子 2 无足够的空间，而箱子 2 的下一个箱子（箱子 1）也无足够的空间，因此启用新箱子——箱子 3，并将物品 4 放入其中。装载物品 5 的过程是从查找箱子 3 的下一个箱子开始的，箱子 3 的下一个箱子为箱子 1，按上述步骤，可查知箱子 2 是合适的，因此将物品 5 放入箱子 2。对于最后一个物品 6，从箱子 3 开始检查，因该箱有足够空间，可将物品 6 放入其中。

上述相邻匹配策略与另一种同名的动态存储分配策略很类似，即每次装载一个物品，若一个物品不能装入当前箱子，则将当前箱子关闭并启用一个新的箱子。本节不准备介绍这种匹配策略。

可用最大赢者树来高效地实现相邻匹配策略。与最先匹配法一样，外部节点代表各箱子，比赛是依据各箱子的最大容量来进行的。对于  $n$  个物品的装载问题，从  $n$  个箱子（外部节点）开始工作。观察图10-7的最大赢者树，其中有6/8的箱子已被使用，各标号的约定同图10-6。虽然当  $n = 8$  时，图10-7所示的情况不会出现，但它演示了如何确定装载下一个物品的箱子。若上一个物品被放入箱子  $last$  中且当前已使用了  $b$  个箱子，则搜索下一个可用的箱子可按如下两个步骤来进行：

1) 找到第一个箱子  $j$  ,  $j > last$ 。当箱子总数为  $n$  时，这样的  $j$  总存在。若该箱子非空（即  $b > 0$ ），则使用之。

2) 若1) 未找到合适的箱子，则在树中搜索适合该物品的最左箱子，这个箱子是目前正在使用的箱子。

现在来考察图10-7中的情形。假设下一个物品需7个单元的空间。若  $last = 3$ ，则在1) 中可确定箱子5有足够的空间。因箱子5是非空箱子，可将物品放入其中。另一方面，若  $last=5$ ，则在1) 中获知箱子7有足够空间，因箱子7为空，故移到2)，找到有足够空间的最左箱子为箱子1，将物品放入其中。

为了实现1)，从箱子  $j=last+1$  开始，其中  $last$  为上一个箱子的编号。注意到若  $last=n$ ，则所有  $n$  个物品都已被装载并且使用了  $n$  个

箱子，每个物品分别放在一个箱子中。因此  $j = n$ 。图10-8的伪代码描述了从箱子  $j$  开始查询合适箱子的过程。一般遍历从箱子  $j$  到根的路径，查询各右子树直到找到第一个含有合适箱子的右子树。当找到该子树时，该子树中具有合适容量的最左箱子就是所查找的箱子。

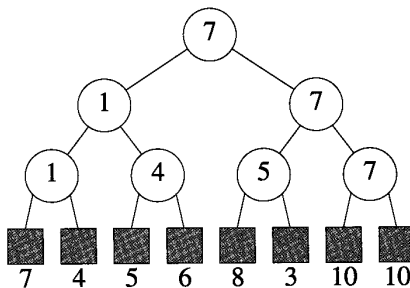


图10-7 相邻匹配法的最大赢者树

```
//寻找距last的右孩子最近的箱子，用来存放物品 i
j = last + 1;
if (avail[j] >= s[i]) return j;
if (avail[j+1] >= s[i]) return j+1;
p = avail[j]的父节点;
if (p == n-1) { // 特殊情况
    令 q 为 t[p]右孩子中的外部节点;
    if (avail[q] >= s[i]) return q;
}

//向树根移动，寻找第一个含有一个有足够容量的箱子的右子树
//p 的右子树为 p+1

p /= 2; //移动到父节点
while (avail[t[p+1]] < s[i])
    p /= 2;

return 子树 p+1中第一个能够存放物品 i 的箱子;
```

图10-8 查找合适箱子的伪代码

考察图 10-7 的赢者树。假设  $last=1$  且  $s[i]=7$ 。从  $j=2$  开始，首先可以确定箱子 2 无足够的容量。接着，查询箱子  $j+1=3$ ，它也没有足够的容量。因此移到  $j$  的父节点并取  $p$  等于 4。因  $p \neq n-1$ ，我们到达 while 循环并得知根为 5 的子树不含合适的箱子。接下来，移到节点 2 并得知根 3 的子树包含合适的箱子。所需的箱子应是该树中容量大于或等于 7 的最左箱子。按程序 10-7 的策略可找到这个箱子——箱子 5。若初始假设为  $last=3$  且  $s[i]=8$ ，则将从箱子 4 开始查询。箱子 4 和箱子 5 都无足够的容量，故取  $p$  等于 5 并到达 while 循环。在第一次循环中检查  $avail[t[6]]$ ，并得知根为 6 的子树不含合适的箱子。然后， $p$  移到节点 2，并得知根为 3 的子树包含合适的箱子。用程序 10-7 的策略可找到这个合适箱子——箱子 7。因箱子 7 是空的，故移到 2)，确定使用箱子 7。

1) 要求我们按树的某条路径向下遍历以找到最左合适箱子，所需的时耗为  $O(\log n)$ 。利用程序 10-7 中的策略，2) 所需的时耗为  $\Theta(\log n)$ 。因此相邻匹配策略总的时间复杂性为  $\Theta(n \log n)$ 。

## 练习

7. 函数 `FirstFitPack` (见程序 10-7) 将一个物品分配给一个箱子的时耗为  $\Theta(\log n)$ ，即使当前所使用的箱子数目远远少于  $n$ 。如果从包含箱子 1 和箱子  $b$  ( $b$  为当前已使用的最右箱子) 的最小子树的根开始搜索，可以进一步减少分配一个箱子的时间，也就是说，从箱子 1 和箱子  $b$  的最近祖先开始搜索。例如，当  $b$  为 3 时，从节点 2 开始搜索。如果在箱子 1 到箱子  $b$  中没有有一个箱子的容量符合条件，则将  $b$  增 1。另外，在重新组织比赛时，只需重赛位于箱子 1 和箱子  $b$  的最近公共祖先之前的比赛。按照上述思想重写程序 10-7，并在  $n=1000$ 、5000、50 000、和 100 000 的情况下，比较前后两个程序版本的时间消耗。

8. 1) 扩充类 `WinnerTree`，增加共享函数 `LeftChild(i)` 和 `RightChild(i)`，函数 `LeftChild(i)` 和 `RightChild(i)` 分别返回内部节点  $i$  的左右孩子，当  $i=0$  时返回值均为 0。

2) 重写 `FirstFitPack` (见程序 10-7)，使之符合 10.5.1 节所述的信息隐藏原理。

9. 虽然证明最先匹配法和最优匹配法的箱子数不会超过  $\lceil (17/10)b(I) \rceil$  很困难 ( $b(I)$  为实例  $I$  下所需的最少箱子数)，但比较容易证明箱子数不会超过  $2b(I)$ ，请证明之。

10. 最差匹配法 (Worst Fit) 是又一种箱子装载策略。同最先匹配法一样，一次只将一个物品放入箱子中。若要装载一个物品，则将其放入可选的具有最大容量的非空箱子中，如果没有这样的箱子则启用一新的箱子来装载此物品。最差匹配法可用最大堆来实现，相应的时间复杂性应为  $O(n \log n)$ ，其中  $n$  为物品的数量。

1) 采用最差匹配法编写一个不同于函数 `FirstFitPack` 的函数 `WorstFitPack`，使用一个初始为空的堆 (即：没有非空箱子)。每次装载一个物品时，搜索具有最大可用空间的箱子，若该箱子无足够的空间则启用一个新箱子并将其加入堆中。

2) 比较在  $n=500$ 、1000、2000 和 5000 的情况下，最差匹配法和最先适配法各自所用的箱子数。

11. 1) 利用 10.5.2 节所述的两个步骤及图 10-8 中的伪代码，编写采用相邻匹配策略实现箱子装载的 C++ 程序。

2) 利用随机产生的箱子装载实例比较相邻匹配策略和最先适配策略所需要的箱子数。

## 第11章 搜索树

本章是关于树结构的最后一章，我们将给出一种适合于描述字典的树形结构。第7章中的字典描述仅提供比较好的平均性能，而在最坏情况下的性能很差。当用跳表来描述一个  $n$  元素的字典时，对其进行搜索、插入或者删除操作所需要的平均时间为  $O(\log n)$ ，而最坏情况下的时间为  $\Theta(n)$ 。当用散列来描述一个  $n$  元素的字典时，对其进行搜索、插入或者删除操作所需要的平均和最坏时间分别为  $\Theta(1)$  和  $\Theta(n)$ 。使用跳表很容易对字典元素进行高效的顺序访问（如按照升序搜索元素），而散列却做不到这一点。当用平衡搜索树来描述一个  $n$  元素的字典时，对其进行搜索、插入或者删除所需要的平均时间和最坏时间均为  $\Theta(\log n)$ ，按元素排名进行的查找和删除操作所需要的时间为  $O(\log n)$ ，并且所有字典元素能够在线性时间内按升序输出。正因为这样（无论是平衡还是非平衡搜索树），所以在搜索树中进行顺序访问时，搜索每个元素所需要的平均时间为  $\Theta(1)$ 。

实际上，如果所期望的操作为查找、插入和删除（均根据元素的关键值来进行），则可以借助于散列函数来实现平衡搜索树。当字典操作仅按关键值来进行时，可将平衡搜索用于那些对时间要求比较严格的应用，以确保任何字典操作所需要的时间都不会超过指定的时间量。平衡搜索树也可用于按排名来进行查找和删除操作的情形。对于那些不按精确的关键值匹配进行字典操作的应用（比如寻找关键值大于  $k$  的最小元素），同样可使用平衡搜索树。

本章将首先介绍二叉搜索树。这种树提供了可与跳表相媲美的渐进复杂性。其搜索、插入和删除操作的平均时间复杂性为  $O(\log n)$ ，最坏时间复杂性为  $\Theta(n)$ 。接下来将介绍两种大家比较熟悉的平衡树：AVL树和红-黑树。无论哪一种树，其搜索、插入和删除操作都能在对数时间内完成（平均和最坏情况）。两种结构的实际运行性能也很接近，AVL树一般稍微快一些。所有的平衡树结构都使用“旋转”来保持平衡。AVL树在执行每个插入操作时最多需要一次旋转，执行每个删除操作时最多需要  $O(\log n)$  次旋转；而红-黑树对于每个插入和删除操作，都需要执行一次旋转。这种差别对于大多数仅需  $\Theta(1)$  时间进行一次旋转的应用来说无关紧要，但对于那些不能在常量时间内完成一次旋转的应用来说就非常重要了，比如平衡优先搜索树 McCreight 就是这样一种应用。平衡优先搜索树用于描述具有两个关键值的元素，此时，每个关键值是一对数  $(x, y)$ 。它同时是一个关于  $y$  的优先队列和关于  $x$  的搜索树。在平衡优先搜索树中执行旋转时，每次旋转都需耗时  $O(\log n)$ 。如果用红-黑树来描述平衡优先搜索树，由于每一次插入或删除后仅需执行一次旋转，因此插入或删除操作总的时间复杂性仍保持为  $O(\log n)$ ；当使用 AVL 树时，删除操作的时间将变为  $O(\log n)$ 。

如果所描述的字典比较小（能够完全放入内存），AVL树和红-黑树均能提供比较高的性能，但对于很大的字典来说，它们就不适用了。当字典存储在磁盘上时，需要使用带有更高次数（因而有更小高度）的搜索树，本章也将介绍一个这样的搜索树——B-树。

本章的应用部分将给出三个搜索树的应用。第一个是直方图的计算，第二个是 10.5.1 节所介绍的 NP-复杂问题——箱子装载，最后一个是关于在电子布线中所出现的交叉分布问题。在直方图的应用中，使用散列函数来取代搜索树，从而使性能得到提高。在最优匹配箱子装载应用中，由于搜索不是按精确匹配完成的，所以不能使用散列函数。在交叉分布问题中，操作是按排名完成的，因此也不能使用散列函数。



## 11.1 二叉搜索树

### 11.1.1 基本概念

7.1和7.4节介绍了抽象数据类型 *Dictionary*，从中可以发现当用散列来描述一个字典时，字典操作（包括插入、搜索和删除）所需要的平均时间为  $\Theta(1)$ 。而这些操作在最坏情况下的时间正比于字典中的元素个数  $n$ 。如果扩充 *Dictionary* 的 ADT 描述，增加以下操作，那么散列将不能再提供比较好的平均性能：

- 1) 按关键值的升序输出字典元素。
- 2) 按升序找到第  $k$  个元素。
- 3) 删除第  $k$  个元素。

为了执行操作 1)，需要从表中取出数据，将它们排序后输出。如果使用除数为  $D$  的链表，那么能在  $\Theta(D+n)$  的时间内取出元素，在  $O(n \log n)$  时间内完成排序和在  $\Theta(n)$  时间内输出，因此共需时间  $O(D+n \log n)$ 。如果对散列使用线性开型寻址，则取出元素所需时间为  $\Theta(b)$ ， $b$  是桶的个数，这时所需时间为  $O(b+n \log n)$ 。如果使用链表，操作 2) 和 3) 可以在  $O(D+n)$  的时间内完成，而如果使用线性开型寻址，它们可在  $\Theta(b)$  时间内完成。为了获得操作 2) 和 3) 的这种复杂性，必须采用一个线性时间算法来确定  $n$  元素集合中的第  $k$  个元素（参考 14.5 节）。

如果使用平衡搜索树，那么对字典的基本操作（搜索、插入和删除）能够在  $O(\log n)$  的时间内完成，操作 1) 能在  $\Theta(n)$  的时间内完成。通过使用带索引的平衡搜索树，也能够在此时间内完成操作 2) 和 3)。11.3 节将考察其他一些散列无法做到而平衡树可以有效解决的应用。

在学习平衡树之前，首先来看一种叫作二叉搜索树的简单结构。

**定义 [二叉搜索树]** 二叉搜索树 (binary search tree) 是一棵可能为空的二叉树，一棵非空的二叉搜索树满足以下特征：

- 1) 每个元素有一个关键值，并且没有任意两个元素有相同的关键值；因此，所有关键值都是唯一的。
- 2) 根节点左子树的关键值（如果有的话）小于根节点的关键值。
- 3) 根节点右子树的关键值（如果有的话）大于根节点的关键值。
- 4) 根节点的左右子树也都是二叉搜索树。

此定义中有一些冗余。特征 2)、3) 和 4) 在一起暗示了关键值必须是唯一的。因此，特征 1) 可以用这样的特征代替：根节点必须有关键值。然而，前一种定义比这种简化的定义要清楚了。

图 11-1 给出了一些各元素含有不同关键值的二叉树。节点中的数字是元素的关键值。其中 11-1a 中的树尽管满足特征 1)、2) 和 3)，但仍然不是二叉搜索树，因为它不满足特征 4)，其中有一个子树的右子树的关键值 (22) 小于该子树根节点的关键值 (25)。而图 11-1b 和 c 都是二叉搜索树。

我们可以放弃二叉搜索树中所有元素拥有不同关键值的要求，然后再用小于等于代替特征 2) 中的小于，用大于等于代替特征 3) 中的大于，这样，就得到了一棵有重复值的二叉搜索树 (binary search tree with duplicates)。

带索引的二叉搜索树 (indexed binary search tree) 源于普通的二叉搜索树，它只是在每个节点中添加一个 LeftSize 域。这个域的值是该节点左子树的元素个数加 1。图 11-2 是两棵带索引

的二叉搜索树。节点里面的数字是元素的关键值，外面的是 LeftSize 的值。注意，LeftSize 同时给出了一个元素在子树中排名。例如，在图 11-2a 的树中，根为 20 的子树中的元素（已排序）分别为 12，15，18，20，25 和 30，根节点的排名为 4（即它在排序后的队列中是第 4 个元素），在根为 25 的子树中的元素（已排序）为 25 和 30，因此 25 的排名为 1 且 LeftSize 的值也为 1。

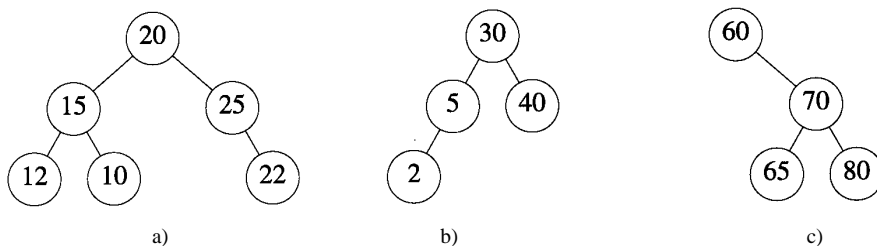


图11-1 二叉树

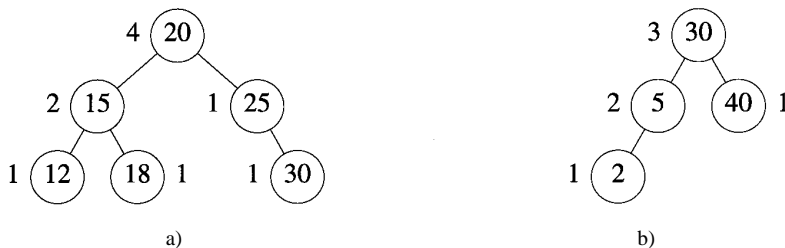


图11-2 带索引的二叉搜索树

### 11.1.2 抽象数据类型 BSTree 和 IndexedBSTree

ADT 11-1 给出了二叉搜索树的抽象数据类型描述。带索引的二叉搜索树支持所有的二叉搜索树操作。另外，它还支持按排名进行的查找和删除操作。ADT 11-2 给出了它的抽象数据类型描述。可以按照类似的方法来描述抽象数据类型 *DBSTree* (有重复值的二叉搜索树) 和 *DIndexedBSTree*。

ADT 11-1 二叉搜索树的抽象数据类型描述

抽象数据类型 *BSTree* {

实例

二叉树，每一个节点中有一个元素，该元素有一个关键值域；所有元素的关键值各不相同；任何节点左子树的关键值小于该节点的关键值；任何节点右子树的关键值大于该节点的关键值。

操作

*Create()*：创建一个空的二叉搜索树

*Search(k, e)*：将关键值为  $k$  的元素返回到  $e$  中；如果操作失败则返回 false，否则返回 true

*Insert(e)*：将元素  $e$  插入到搜索树中

*Delete(k, e)*：删除关键值为  $k$  的元素并且将其返回到  $e$  中

*Ascend()*：按照关键值的升序排列输出所有元素

}



## ADT 11-2 带索引的二叉搜索树的抽象数据类型描述

抽象数据类型 *IndexedBSTree* {

实例

除每一个节点有一个 *LeftSize* 域以外，其他与 *BSTree* 相同

操作

*Create()*：产生一个空的带索引的二叉搜索树

*Search(k,e)*：将关键值为 *k* 的元素返回到 *e* 中；如果操作失败返回 *false*，否则返回 *true*

*IndexSearch(k,e)*：将第 *k* 个元素返回到 *e* 中

*Insert(e)*：将元素 *e* 插入到搜索树

*Delete(k,e)*：删除关键值为 *k* 的元素并且将其返回到 *e* 中

*IndexDelete(k,e)*：删除第 *k* 个元素并将其返回到 *e* 中

*Ascend()*：按照关键值的升序排列输出所有元素

}

11.1.3 类 *BSTree*

因为在执行操作时，二叉搜索树中元素的数量和树的外形同时改变，所以可以用 8.4 节中的链表来描述二叉搜索树。如果从类 *BinaryTree*（见程序 8-7）中派生类 *BSTree*，那么可以大大简化 *BSTree* 类的设计，见程序 11-1。由于 *BSTree* 是从 *BinaryTree* 派生而来的，因此它继承了 *BinaryTree* 的所有成员。但是，它只能访问那些共享成员和保护成员。为了访问 *BinaryTree* 私有成员 *root*，需要把 *BSTree* 定义为 *BinaryTree* 的友元。

程序 11-1 二叉搜索树的类定义

```
template<class E, class K>
class BSTree : public BinaryTree<E> {
public:
    bool Search(const K& k, E& e) const;
    BSTree<E,K>& Insert(const E& e);
    BSTree<E,K>& Delete(const K& k, E& e);
    void Ascend() {InOutput();}
};
```

*IndexedBSTree* 类也可以定义为 *BinaryTree* 的一个派生类（见练习 5）。可以通过调用 8.9 节所定义的中序输出函数——*InOutput* 将二叉搜索树按升序输出，该函数首先输出左子树中的元素（关键值较小的元素），然后输出根，最后输出右子树中的元素（关键值较大的元素）。对于有 *n* 个元素的树来说，该函数的时间复杂性为  $\Theta(n)$ 。

## 11.1.4 搜索

假设需要查找关键值为 *k* 的元素，那么先从根开始。如果根为空，那么搜索树不包含任何元素，查找失败，否则，将 *k* 与根的关键值相比较，如果 *k* 小于根节点的关键值，那么就不必搜索右子树中的元素，只要在左子树中搜索即可。如果 *k* 大于根节点的关键值，则正好相反，只需在右子树中搜索即可。如果 *k* 等于根节点的关键值，则查找成功，搜索终止。在

子树中的查找与此类似，程序 11-2 给出了相应代码。该过程的时间复杂性为  $O(h)$ ，其中  $h$  是树的高度。

程序11-2 在二叉搜索树中搜索元素

```
template<class E, class K>
bool BSTree<E,K>::Search(const K& k, E &e) const
{
    // 搜索与k匹配的元素
    // 指针 p 从树根开始进行查找
    BinaryTreeNode<E> *p = root;
    while (p) // 检查p->data
        if (k < p->data) p = p->LeftChild;
        else if (k > p->data) p = p->RightChild;
        else { // 找到元素
            e = p->data;
            return true;
        }
    return false;
}
```

可以用类似的方法在带索引的二叉搜索树中按索引进行查找。假设需要查找图 11-2a 中树的第三个元素，根节点的 LeftSize 为 4，因此第三个元素在左子树中。左子树根节点的 LeftSize 为 2，因此第三个元素是左子树的右子树中的最小元素，而右子树根节点的 LeftSize 是 1，所以此根节点就是要找的元素。该操作时间复杂性也是  $O(h)$ 。

### 11.1.5 插入

若在二叉搜索树中插入一个新元素  $e$ ，首先要验证  $e$  的关键值与树中已有元素的关键值是否相同，这可以通过用  $e$  的关键值对二叉树进行搜索来实现。如果搜索不成功，那么新元素将被插入到搜索的中断点。例如，要将关键值为 80 的元素插入到图 11-1b 所示的树中去，首先对 80 进行搜索，由于搜索不成功而中断，最后检验的节点是关键值为 40 的节点，新元素将被插入到该节点之下作为其右孩子。插入后的结果如图 11-3a 所示。图 11-3b 给出了将关键值为 35 的元素插入到图 11-3a 所示二叉树之后的结果。程序 11-3 实现了上述插入策略。

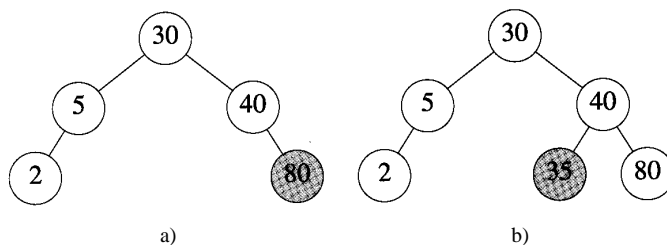


图11-3 将新元素插入到二叉搜索树中

程序11-3 将元素插入到二叉搜索树中

```
template<class E, class K>
BSTree<E,K>& BSTree<E,K>::Insert(const E& e)
{
    // 如果不出现重复，则插入 e
}
```

```

BinaryTreeNode<E> *p = root, // 搜索指针
                    *pp = 0; // p的父节点指针

// 寻找插入点
while (p) { // 检查 p->data
    pp = p;
    // 将p移向孩子节点
    if (e < p->data) p = p->LeftChild;
    else if (e > p->data) p = p->RightChild;
    else throw BadInput(); // 出现重复
}

// 为e 建立一个节点，并将该节点连接至 to pp
BinaryTreeNode<E> *r = new BinaryTreeNode<E> (e);
if (root) { // 树非空
    if (e < pp->data) pp->LeftChild = r;
    else pp->RightChild = r;}
else // 插入到空树中
    root = r;

return *this;
}

```

当将元素插入到带索引的二叉搜索树中时，可以用一个与程序 11-3 相类似的过程，但此时需要更新从根节点到新插入节点路径上的所有节点的 LeftSize 值。插入过程仍然可以在  $O(h)$  时间内完成，其中  $h$  是树的高度。

### 11.1.6 删除

对删除来说，我们考虑包含被删除元素的节点  $p$  的三种情况：1)  $p$  是树叶；2)  $p$  只有一个非空子树；3)  $p$  有两个非空子树。

情况1) 可以用丢弃树叶节点的方法来处理。要删除图 11-3b 所示树中的 35，只要把其父节点的左孩子域置为零，然后删除该节点即可，删除后结果如图 11-3a 所示。要从树中删除 80，只要把节点 40 的右孩子域置为零并丢弃节点 80，其结果如图 11-1b 所示。

接下来考察情况2)。如果  $p$  没有父节点（即  $p$  是根节点），则将  $p$  丢弃， $p$  的唯一子树的根节点成为新的搜索树的根节点。如果  $p$  有父节点  $pp$ ，则修改  $pp$  的指针，使得  $pp$  指向  $p$  的唯一孩子，然后删除节点  $p$ 。例如，如果希望从图 11-3b 的树中删除关键值为 5 的元素，则修改该元素父节点的左孩子域，使其指向关键值为 2 的节点。

最后，要删除一个左右子树都不为空的节点中的元素，只需将该元素替换为它的左子树中的最大元素或右子树中的最小元素。假设希望删除图 11-4a 中关键值为 40 的元素，那么既可以用它左子树中的最大元素（35），也可以用它右子树中的最小元素（60）来替换它。如果选择右子树中的最小元素，那么把关键值为 60 的元素移到 40 被删除的位置，再把原来的叶节点 60 删除即可。结果如图 11-4b 所示。

假定用左子树中的最大元素来代替被删除的元素 40。左子树中的最大元素是 35，且只有一个子女，把 35 移到 40 的节点中，将其左孩子指向原来节点 35 的唯一子女，结果如图 11-4c 所示。

再来看另一个例子，删除图 11-4c 中的节点 30。既可以用 5，也可以用 31 来替换节点 30。如果选用 5，而 5 是只有一个孩子的节点，那么只要把其左孩子域指向 5 原来的唯一子女即可，结

果如图11-4d所示。如果选用31替换30，而原来的31是树叶节点，那么只需删除该树叶节点。

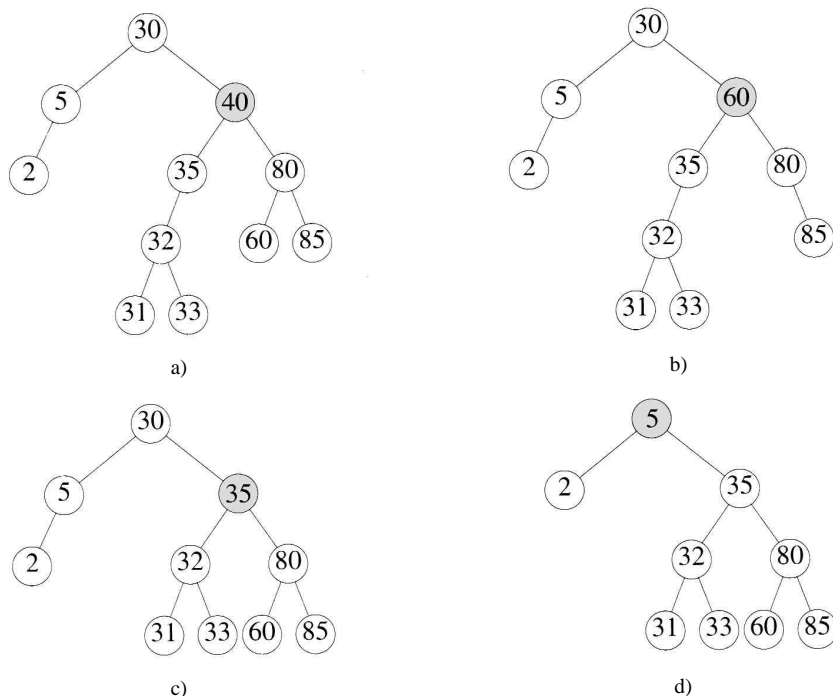


图11-4 二叉搜索树中元素的删除

注意，必须确保右子树中的最小元素以及左子树中的最大元素既不会在没有子树的节点中，也不会只有一个子树的节点中。可以按下述方法来查找左子树中的最大元素：首先移动到子树的根，然后沿着各节点的右孩子指针移动，直到右孩子指针为0为止。类似地，也可以找到右子树中的最小元素：首先移动到子树的根，然后沿着各节点的左孩子指针移动，直到左孩子指针为0为止。

程序11-4给出了上述删除操作的算法。删除一个有两个非空子树的节点时，该程序一般使用左子树的最大元素来进行替换，其复杂度为  $O(h)$ 。用类似的过程，可以在相同的时间内完成带索引二叉搜索树的删除操作。首先按索引进行搜索，找到被删除元素，然后将其删除，如果需要的话，还需要修改从根至被删除元素路径上所有节点的 LeftSize 域。

程序11-4 二叉搜索树的删除

```
template<class E, class K>
BSTree<E,K>& BSTree<E,K>::Delete(const K& k, E& e)
// 删除关键值为 k 的元素，并将其放入 e

// 将 p 指向关键值为 k 的节点
BinaryTreeNode<E> *p = root, // 搜索指针
                *pp = 0; // p 的父节点指针
while (p && p->data != k){ // 移动到 p 的孩子
    pp = p;
    if (k < p->data) p = p->LeftChild;
```

```

    else p = p->RightChild;
}
if (!p) throw BadInput(); // 没有关键值为k的元素

e = p->data; // 保存欲删除的元素

// 对树进行重构
// 处理p有两个孩子的情形
if (p->LeftChild && p->RightChild) { // 两个孩子
    // 转换成有0或1个孩子的情形
    // 在 p 的左子树中寻找最大元素
    BinaryTreeNode<E> *s = p->LeftChild, *ps = p; // s的父节点
    while (s->RightChild) { // 移动到较大的元素
        ps = s;
        s = s->RightChild;
    }

    // 将最大元素从s移动到p
    p->data = s->data;
    p = s;
    pp = ps;
}

// p 最多有一个孩子
// 在 c 中保存孩子指针
BinaryTreeNode<E> *c;
if (p->LeftChild) c = p->LeftChild;
else c = p->RightChild;

// 删除p
if (p == root) root = c;
else { // p 是 pp的左孩子还是pp的右孩子?
    if (p == pp->LeftChild)
        pp->LeftChild = c;
    else pp->RightChild = c;
}
delete p;

return *this;
}

```

### 11.1.7 类DBSTree

若二叉搜索树中的不同元素可以包含相同的关键值，则称这种树为 DBSTree。在实现 DBSTree类时，只需把 BSTree::Insert的while循环（见程序 11-3）改为程序 11-5所示的while循环即可，其他代码无须改动。

程序11-5 对程序 11-3的while 循环进行修改

```

while (p) {
    pp = p;
    if (e <= p->data) p = p->LeftChild;
}

```

```
else p = p->RightChild;  
}
```

### 11.1.8 二叉搜索树的高度

一棵  $n$  元素的二叉搜索树的高度可以与  $n$  一样大。例如，用程序 11-3 将一组关键值为  $[1, 2, 3, \dots, n]$  的元素按顺序插入到一棵空的二叉搜索树时，树的高度就会这样大，对树的搜索、插入和删除操作所需要的时间均为  $O(n)$ ，这不比那些使用无序链表的操作好多少。但是可以证明，当用程序 11-3 和程序 11-4 进行随机插入和删除操作时，二叉搜索树的平均高度是  $O(\log n)$ 。因此，每一个树操作的平均时间是  $O(\log n)$ 。

### 练习

1. 用跳表实现 ADT11-1 中的 BSTree 操作需要多少时间（平均性能）？
2. 请描述抽象数据类型 DBSTree（有重复值的二叉搜索树）。
3. 请描述抽象数据类型 DIndexedBSTree。
4. 从 BSTree 类（见程序 11-1）中派生一个 C++ 类 DBSTree，要求各函数的复杂性应与 BSTree 相同。测试程序的正确性。
5. 从 BSTree 类（见程序 11-1）中派生一个 C++ 类 IndexedBSTree。可以假设 LeftSize 域是 data 域的一个子域。检验程序的正确性。根据元素的个数和（或）树的高度给出每个函数的复杂性。
6. 设计一个类 IndexedBinaryTree，用于把线性表描述成一个二叉树（非二叉搜索树）。类必须支持程序 3-1 中所定义的所有线性表操作。除了 Search 函数以外，其他操作都必须在对数时间或更少的时间内完成。可以假设二叉树的平均高度是元素个数的对数。
7. 用 DIndexedBSTree 替换 IndexedBSTree 完成练习 5，此时要求从 DBSTree 类中派生 DIndexedBSTree 类。
8. 首先产生一个从 1 到  $n$  的随机排列，然后将关键值为 1 到  $n$  的元素按照随机产生的排列顺序插入到一棵空的二叉树搜索树中。试测量二叉树的高度。重复以上实验，计算测量高度的平均值并与  $2\lceil \log_2(n+1) \rceil$  比较。 $n$  的值可分别取 100、500、1000、10 000、20 000、50 000。
9. 将程序 11-2 中 while 循环的第一次比较改为  $k == p \rightarrow data$ ，再用练习 8 中的方法随机产生不同大小的搜索树，比较修改前和修改后的程序在搜索树中元素时所需要的时间，从中能得出什么结论？
10. 二叉搜索树可用来对  $n$  个元素进行排序。编写一个排序过程，首先将  $n$  个元素  $a[1:n]$  插入到一棵空的二叉搜索树中，然后对树进行中序遍历，并将元素按序放入数组  $a$  中。为简单起见，假设  $a$  中的数是互不相同的。将此过程的平均运行时间与插入排序和堆排序进行比较。
11. 编写一个从二叉搜索树中删除最大元素的函数，函数的时间复杂性必须是  $O(h)$ ，其中  $h$  是二叉搜索树的高度。
  - 1) 用合适的测试数据测试代码的正确性。
  - 2) 随机产生一个  $n$  个元素的线性表和一个长度为  $m$  的插入和最大删除操作序列。在所产生的操作序列中，插入操作的出现概率应近似为 0.5（同样，最大删除操作的概率也近似为 0.5）。使用第一个随机线性表中的  $n$  个元素初始化一个最大堆和一棵二叉搜索树。检测用堆和二叉搜索树执行  $m$  个操作的时间，用该时间除以  $m$  就得到每一操作的平均时间。重复进行此实验，取

$n=100, 500, 1000, 2000, \dots, 5000$ ，假设 $m=5000$ 。将所得结果用表格形式给出。

3) 指出这两种优先队列的相对优点和缺点。

\*12. 扩充BinarySearchTree类，增加两个顺序访问函数Begin和Next。这两个函数分别返回指向字典第一个元素和下一个元素的指针。若没有第一个元素或没有下一个元素时，它们的返回值都是零。试证明这两个函数的平均复杂性均为 $O(1)$ 。测试代码的正确性。

## 11.2 AVL树

### 11.2.1 基本概念

当确定搜索树的高度总是 $O(\log n)$ 时，能够保证每个搜索树操作所占用的时间为 $O(\log n)$ 。高度为 $O(\log n)$ 的树称为平衡树（balanced tree）。1962年，Adelson-Velskii 和Landis 提出了一种现在非常流行的平衡树——AVL树（AVL tree）。

**定义** 空二叉树是AVL树；如果 $T$ 是一棵非空的二叉树， $T_L$ 和 $T_R$ 分别是其左子树和右子树，那么当 $T$ 满足以下条件时， $T$ 是一棵AVL树：1)  $T_L$ 和 $T_R$ 是AVL树；2)  $|h_L - h_R| \leq 1$ ， $h_L$ 和 $h_R$ 分别是左子树和右子树的高度。

AVL搜索树既是二叉搜索树，也是AVL树，图11-1a和b中的树都是AVL树，而c不是。树a不是AVL搜索树，因为它不是二叉搜索树。树b是AVL搜索树，图11-3中的树也是AVL搜索树。

带索引的AVL搜索树既是带索引的二叉搜索树，也是AVL树。图11-2中的搜索树都是带索引的AVL搜索树，本节将不再具体介绍带索引的AVL搜索树。

如果用AVL树来描述字典并希望对数时间内完成每一种字典操作，那么，AVL树必须具备下述特征：

- 1)  $n$  个元素（节点）的AVL树的高度是 $O(\log n)$ 。
- 2) 对于每一个 $n$  ( $n > 0$ ) 值，都存在一棵AVL树。（否则，在插入完成后，一棵AVL树将不再是AVL树，因为对当前元素数来说不存在对应的AVL树）
- 3) 一棵 $n$  元素的AVL搜索树能在 $O(\text{高度})=O(\log n)$  的时间内完成搜索。
- 4) 将一个新元素插入到一棵 $n$  元素的AVL搜索树中，可得到一棵 $n+1$ 元素的AVL树，这种插入过程可以在 $O(\log n)$ 时间内完成。
- 5) 从一棵 $n$  元素的AVL搜索树中删除一个元素，可得到一棵 $n-1$ 元素的AVL树，这种删除过程可以在 $O(\log n)$ 时间内完成。

特征4) 包含了特征2)，因此不需要明确说明特征2)，特征1)、3)、4) 和5) 将在以下小节中详细介绍。

### 11.2.2 AVL树的高度

我们能够获得一棵 $n$  节点的AVL树的高度的范围。假设 $N_h$  是一棵高度为 $h$  的AVL树中最小的节点数。在最坏情况下，根节点的两个左右子树中一棵子树的高度是 $h-1$ ，另一棵子树的高度是 $h-2$ ，而且两棵子树都是AVL树。因此有：

$$N_h = N_{h-1} + N_{h-2} + 1, N_0 = 0, N_1 = 1$$

可以看到 $N_h$  的定义与斐波那契数列的定义非常相似：

$$F_n = F_{n-1} + F_{n-2}, F_0 = 0, F_1 = 1$$



也可以这样来表示： $N_h = F_{h+2} - 1$ ,  $h \geq 0$  (见练习11)。由斐波那契定理可以知道  $F_h \approx \phi^h / 5$ , 其中  $\phi = (1 + \sqrt{5})/2$ , 因此  $N_h \approx \phi^{h+2} / 5 - 1$ 。如果树中有  $n$  个节点, 那么树的最大高度为:  $\log_\phi (5(n+1)) - 2 \sim 1.44 \log_2 (n+2) = O(\log n)$ 。

### 11.2.3 AVL树的描述

一般用链表方式来描述AVL树, 但是, 为简化插入和删除操作, 我们为每个节点增加一个平衡因子  $bf$ 。节点  $x$  的平衡因子  $bf(x)$  定义为:

$x$  的左子树的高度 -  $x$  的右子树的高度

从AVL树的定义可以知道, 平衡因子的可能取值为 -1, 0和1。图11-5给出了两棵AVL搜索树和树中每个节点的平衡因子。

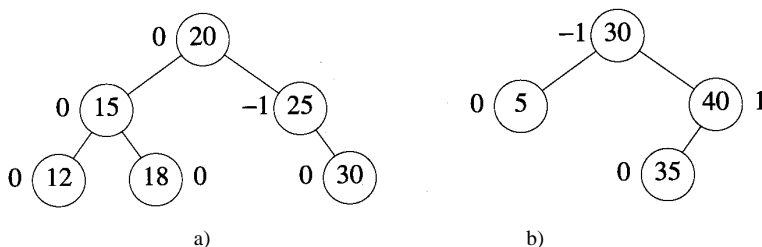


图11-5 AVL搜索树。每个节点外侧的数字是该节点的平衡因子

### 11.2.4 AVL搜索树的搜索

程序11-2可以不作任何修改就用于AVL搜索树的搜索。因为  $n$  元素AVL树的高度是  $O(\log n)$ , 所以搜索所需时间为  $O(n \log n)$ 。

### 11.2.5 AVL搜索树的插入

如果用程序11-3的方法将元素插入到AVL搜索树中, 得到的树可能不再是AVL树。例如, 如把一个关键值为32的元素插入到图11-5b的AVL树中时, 得到的新的搜索树如图11-6a所示。由于新树的节点中所包含的平衡因子不是 -1, 0和1, 所以新树不是AVL树。当用程序11-3的方法将一个新元素插入到AVL树中时, 若得到的新树中有一个或多个节点的平衡因子的值不是 -1, 0或1, 那么就说新树是不平衡的。可以通过移动不平衡树的子树来恢复树的平衡。

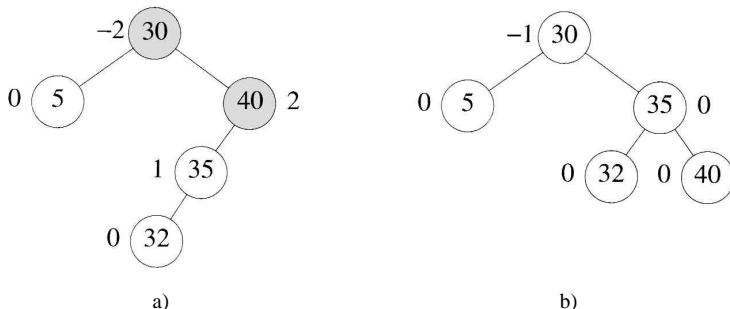


图11-6 向AVL搜索树中插入元素

a) 刚插入时 b) 重新平衡之后

为了恢复平衡,在检查需移动的子树之前,先观察一下由插入操作导致产生不平衡树的几种现象:

- 1) 不平衡树中的平衡因子的值限于-2, -1, 0, 1和2。
- 2) 平衡因子为2的节点在插入前平衡因子为1,与此类似,平衡因子为-2的,插入前为-1。
- 3) 从根到新插入节点的路径上,只有经过的节点的平衡因子在插入后会改变。
- 4) 假设 $A$ 是新插入节点最近的祖先,它的平衡因子是-2或2(图11-6a的例子中, $A$ 是关键值为40的节点),那么,在插入前从 $A$ 到新插入节点的路径上,所有节点的平衡因子都是0。

当我们从根节点往下移动寻找插入新元素的位置时,能够确定节点 $A$ 。从2)中可以知道 $bf(A)$ 在插入前的值既可以是-1,也可以是1。设 $X$ 是最后一个具有这样平衡因子的节点,当把32插入到图11-5b的AVL树中时, $X$ 是关键值为40的节点;当把22, 28或50插入到图11-5a的AVL树中时, $X$ 是关键值为25的节点;当把10, 14, 16或19插入到图11-5a的AVL树中时,这样的节点 $X$ 不存在。

如果节点 $X$ 不存在,那么从根节点至新插入节点途中经过的所有节点在插入前的平衡因子值都是0。由于插入操作只会使平衡因子增/减-1, 0或1,并且只有从根节点至新插入节点途中经过的节点的平衡因子值才会被改变,所以插入后,树的平衡不会被破坏。因此,如果插入后的树是不平衡的,那么 $X$ 就一定存在。如果插入后 $bf(X)=0$ ,那么以 $X$ 为根节点的子树的高度在插入前后是相同的。例如,如果插入前的高度是 $h$ ,且 $bf(X)$ 为1,那么,在插入前, $X$ 的左子树的高度 $X_L$ 是 $h-1$ ,右子树的高度 $X_R$ 是 $h-2$ (如图11-7a所示)。由于平衡因子变为0,所以必须在 $X_R$ 中作插入,得到高度为 $h-1$ 的新子树 $X'_R$ (如图11-7b所示)。由于从 $X$ 到新插入节点途中遇到的所有节点在插入前的平衡因子均为0,所以 $X'_R$ 的高度必须增加到 $h-1$ 。 $X$ 的高度仍保持为 $h$ , $X$ 的祖先的平衡因子在插入前后保持相同,这样,所以树的平衡被保持住了。

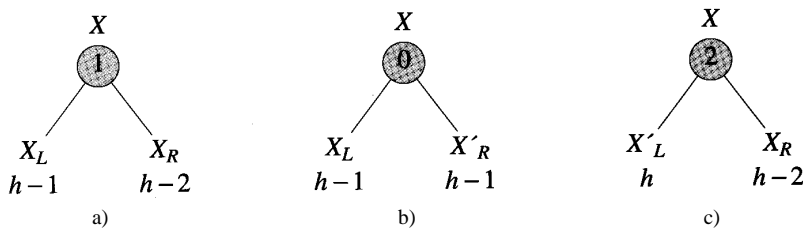


图11-7 向AVL搜索树中插入元素

a) 插入之前 b) 插入到 $X_R$ 之后 c) 插入到 $X_L$ 之后

使树的平衡遭到破坏的唯一一种情形是插入过程使得平衡因子 $bf(X)$ 的值由-1变为-2,或者由1变为2。对于后一种情况来说,插入操作一定是在 $X$ 的左子树 $X_L$ 中进行(如图11-7c所示)的。现在要把 $X'_L$ 的高度调整为 $h$ (因为在插入前所有从 $X$ 到新插入节点途中的节点的平衡因子都为0),因此, $X$ 即4)中所说的节点 $A$ 。

当节点 $A$ 已经被确定时, $A$ 的不平衡性可归类为 $L$ 型不平衡(新插入节点在 $A$ 的左子树中)或 $R$ 型不平衡。通过确定 $A$ 的哪一个孙节点在通往新插入节点的路径上,可以进一步细分不平衡类型。注意这种孙节点肯定存在,这是因为 $A$ 节点的平衡因子是-2或2,所以节点 $A$ 包含新插入节点的子树的高度至少必须是2。根据上述细分不平衡类型的方法, $A$ 节点的不平衡类型将是LL(新插入节点在 $A$ 节点的左子树的左子树中),LR(新插入节点在 $A$ 节点的左子树的右子树中),RR和RL四种类型中的一种。

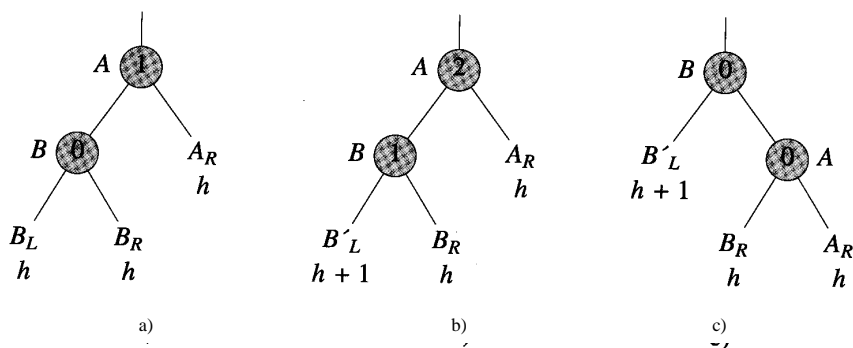
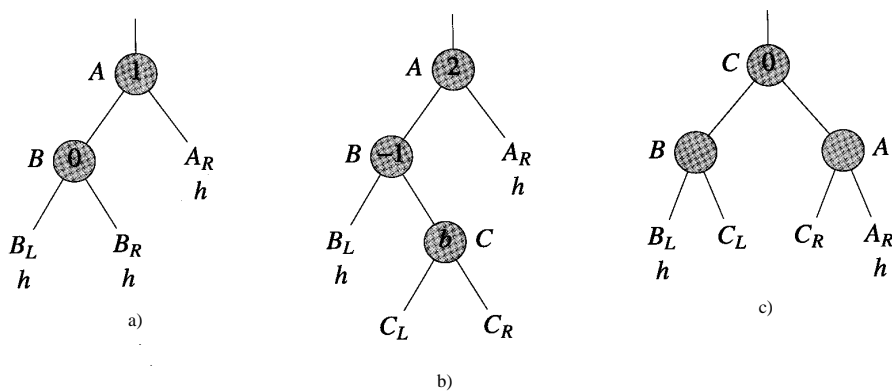


图11-8 LL旋转。节点内为平衡因子，子树名称下面为子数的高度

a) 插入之前 b) 插入 $B_L$ 之后 c) LL旋转之后

图11-8显示了一种普通的LL型不平衡。图11-8a 给出了插入前的条件，图11-8b 是在节点 $B$ 的左子树 $B'$ 中插入一个元素后的情形，而恢复平衡所进行的子树移动如图11-8c所示。原来以 $A$ 为根节点的子树，现在以 $B$ 为根节点， $B'_L$ 仍然是 $B$ 的左子树， $A$ 变成 $B$ 的右子树， $B_R$ 变成 $A$ 的左子树， $A$ 的右子树不变。由于 $A$ 的平衡因子改变了，所以处于从 $B$ 到新插入节点途中的 $B'$ 的所有节点的平衡因子都将改变，其他节点的平衡因子与旋转前保持一致。图11-8中a和c子树的高度是一样的，所以，子树的祖父节点的平衡因子与插入前是一样的。因此不再有平衡因子不是-1, 0或1的节点。一个LL旋转就已经使整个树重新获得平衡！可以验证重新平衡后的树确实是一棵二叉搜索树。



若 $b=0$ ，则 $bf(B)=bf(A)=0$   
 若 $b=1$ ，则 $bf(B)=0$ ， $bf(A)=-1$   
 若 $b=-1$ ，则 $bf(B)=1$ ， $bf(A)=0$

图11-9  $L_R$ 旋转a) 插入之前 b) 插入 $B_R$ 之后 c) LR旋转之后

图11-9给出了一种普通的LR型不平衡。因为插入操作发生在 $B$ 的右子树，这个子树在插入后不可能为空，因此 $C$ 是存在的。但是，它的子树 $C_L$ 和 $C_R$ 有可能为空。为了恢复平衡，需要对子树进行重新整理，如图11-9c所示。重新整理后， $bf(B)$ 和 $bf(A)$ 的值取决于 $bf(C)$ 在插入之后、重新整理之前的值 $b$ 。可以看到，重新整理后的子树仍是二叉搜索树。另外，由于图

11-9a 和 c 中子树的高度是相同的, 所以它们祖先 (如果有的话) 的平衡因子在插入前与在插入后也是相同的。因此, 一个 LR 旋转即可完成整个树的平衡。

RR 和 RL 与上面所讨论的情形是对称的。我们把矫正 LL 和 RR 型不平衡所作的转换称为单旋转 (single rotation), 而把矫正 LR 和 RL 型不平衡所作的转换称为双旋转 (double rotation)。对 LR 型不平衡的转换可以看作 RR 旋转后的 LL 旋转, 而对 RL 型不平衡的转换可以看作 LL 旋转后的 RR 旋转 (练习 15)。

根据上述讨论, 可得到 AVL 搜索树的插入算法, 其步骤如图 11-10 所示。这些步骤可以用 C++ 代码重写, 其复杂性为  $O(\text{高度})=O(\log n)$ 。注意, 如果插入引起了不平衡, 使用单旋转就足以恢复平衡。

- 1) 沿着从根节点开始的路径对具有相同关键值的元素进行搜索, 以找到插入新元素的位置。在此过程中, 寻找最近的, 平衡因子为 -1 或 1 的节点, 令其为 A 节点。如果找到了相同关键值的元素, 那么插入失败, 以下步骤无需执行。
- 2) 如果没有这样的节点 A, 那么从根节点开始再遍历一次, 并修改平衡因子, 然后终止。
- 3) 如果  $bf(A)=1$  并且新节点插入到 A 的右子树中, 或者  $bf(A)=-1$  并且插入是在左子树中进行的, 那么 A 的新平衡因子是 0。这种情况下, 修改从 A 到新节点途中的平衡因子, 然后终止。
- 4) 确定 A 的不平衡类型并执行相应的旋转, 在从新子树根节点至新插入节点途中, 根据旋转需要修改相应的平衡因子。

图11-10 AVL搜索树的插入步骤

### 11.2.6 AVL搜索树的删除

通过执行程序 11-4, 可从 AVL 搜索树中删除一个元素。设  $q$  是被删除节点的父节点。如果要删除图 11-5a 树中关键值为 25 的元素, 那么删除包含该元素的节点, 并且将根节点的右孩子指针指向被删除节点的唯一孩子。根节点是被删除节点的父节点, 所以  $q$  就是根节点。如果被删除元素的关键值是 15, 那么关键值为 12 的元素将占用它的位置, 而原来包含此元素的节点被删除。现在  $q$  是原先包含 15 的节点 (根的左孩子)。由于从根到  $q$  途中的一些 (全部) 节点的平衡因子随着删除操作而改变了, 所以再从  $q$  沿原路返回根节点。

如果删除发生在  $q$  的左子树, 那么  $bf(q)$  减 1, 而如果删除发生在  $q$  的右子树, 那么  $bf(q)$  加 1。可以看到如下现象:

- 1) 如果  $q$  新的平衡因子是 0, 那么它的高度减少了 1, 并且需要改变它的父节点 (如果有的话) 和其他某些祖先节点的平衡因子。
- 2) 如果  $q$  新的平衡因子是 -1 或 1, 那么它的高度与删除前相同, 并且无需改变其祖先的平衡因子值。
- 3) 如果  $q$  新的平衡因子是 -2 或 2, 那么树在  $q$  节点是不平衡的。

由于平衡因子可以沿从  $q$  到根节点的路径改变 (见 2)), 所以途中节点的平衡因子有可能为 2 或 -2。设 A 是第一个这样的节点, 若要恢复 A 节点的平衡, 需要确定其不平衡的类型。如果删除发生在 A 的左子树, 那么不平衡是 L 型; 否则, 不平衡就是 R 型。如果删除后  $bf(A)=2$ , 那么在删除之前  $bf(A)$  的值一定为 1。因此, A 有一棵以 B 为根的左子树。根据  $bf(B)$  的值, 可以把一个 R 型不平衡细分为 R0, R1 和 R-1 类型。例如, R-1 类型指的是这种情况: 删除操作发生在 A 的右子树并且  $bf(B)=-1$ 。类似的, L 型不平衡也可以细分为 L0, L1 和 L-1 类型。

可通过旋转来矫正 A 点的 R0 型不平衡, 如图 11-11 所示。注意图中子树的高度在删除前和删除后都是  $h+2$ , 因此, 到根节点途中的其他节点的平衡因子值没有改变。所以, 整棵树重新获得了平衡。

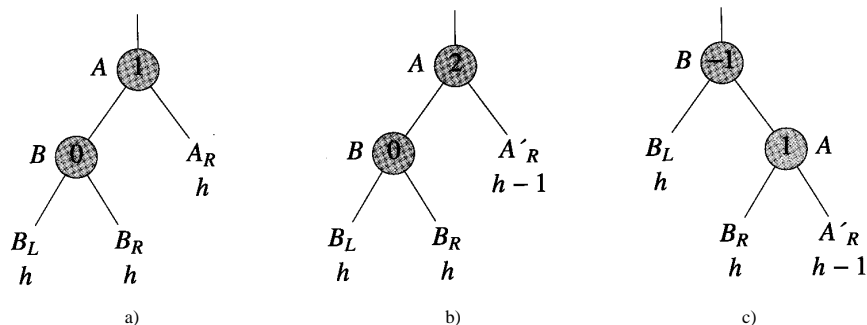


图 11-11 R0 类型的旋转 (单旋转)

a) 删除之前 b) 以  $A_R$  中删除之后 c) R0 旋转之后

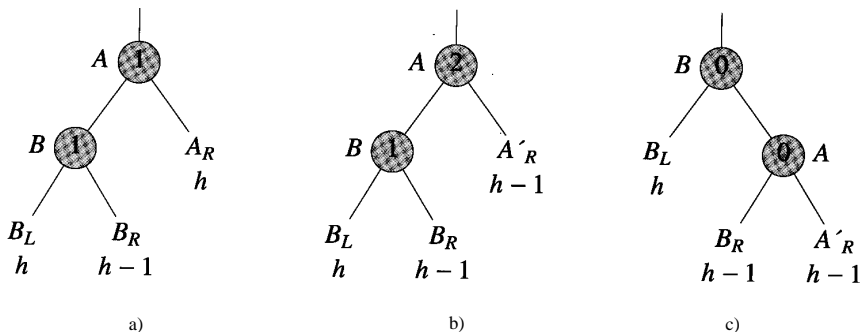
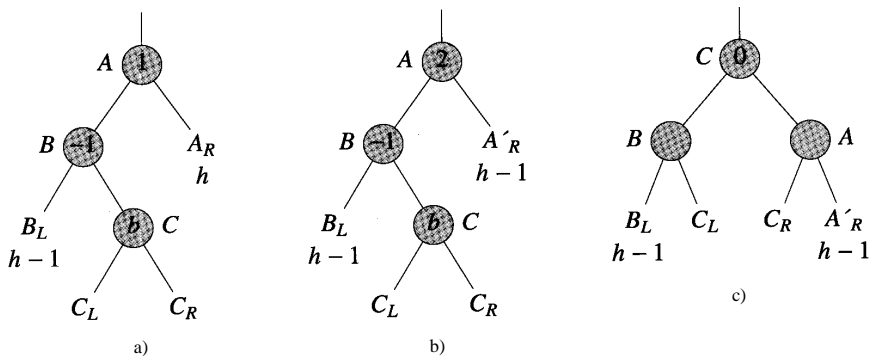


图 11-12 R1 类型的旋转 (单旋转)

a) 删除之前 b) 以  $A_R$  中删除之后 c) R1 旋转之后



若  $b=0$ , 则  $bf(A)=bf(B)=0$   
 若  $b=1$ , 则  $bf(A)=-1$ ,  $bf(B)=0$   
 若  $b=-1$ , 则  $bf(A)=0$ ,  $bf(B)=1$

图 11-13 R-1 类型的旋转 (双旋转)

a) 删除之前 b) 以  $A_R$  中删除之后 c) R-1 旋转之后

图11-12给出了如何处理R1型不平衡。当指针的变化与R0型不平衡中的变化相同时， $A$ 和 $B$ 的新平衡因子是不相同的并且旋转后子树的高度将是 $h+1$ ，此高度比删除操作前减少了1。因此，如果 $A$ 不是根节点，它的某些祖先的平衡因子将产生变化，可能需要进行旋转以保持平衡。R1旋转后，必须继续检查到达根节点路径上的节点。与插入情况不同，在删除操作之后，一次旋转可能还无法恢复平衡。所需要的旋转次数为 $O(\log n)$ 。

R-1类型的不平衡所需要的转换如图11-13所示。节点 $A$ 和 $B$ 旋转后的平衡因子取决于 $B$ 的右孩子的平衡因子 $b$ ，这次旋转得到了一棵高度为 $h+1$ 的子树，而删除前子树的高度是 $h+2$ ，因此，需要在到达根节点的路径上继续旋转。

LL与R1类型的旋转相同；LL与R0型旋转的区别仅在于 $A$ 和 $B$ 最后的平衡因子；而LR与R-1旋转也完全相同。

## 练习

13. 用数学归纳法证明：一棵高度为 $h$ 的AVL树的最少节点数是

$$N_h = F_{h+2} - 1, h \geq 0$$

14. 用程序11-3的方法证明11.2.5节中由插入操作导致不平衡树的现象1)~4)。

15. 为插入前 $bf(X) = -1$ 这种情况画一个类似于图11-7的图。

16. 为RR和RL不平衡画一个类似于图11-8和11-9的图。

17. 从图11-9b所示的LR不平衡开始，画一个在 $B$ 节点执行一个RR旋转的结果示意图。注意，对得到的树执行一次LL旋转即可得到图11-9b的树。

18. 为L0，L1和L-1不平衡情况分别画一个类似于图11-11，11-12和11-13的图。

\*19. 设计一个C++的类AVLtree，它包含二叉搜索树的Search, Insert, Delete 和Ascend 函数，给出所有函数的代码并检验其正确性。前三种函数的时间复杂性应为 $O(\log n)$ ，最后一种函数的时间复杂性应为 $\Theta(n)$ 。

\*20. 针对如下情况完成练习19：二叉搜索树中有一些元素的关键值相同。新类的名称为DAVLtree。

\*21. 设计一个C++类IndexedAVLTree，其中包含索引二叉搜索树的如下函数：Search, Insert, Delete, IndexSearch, IndexDelete和Ascend。给出函数的全部代码并检验其正确性。前5种函数的时间复杂性应为 $O(\log n)$ ，最后一种函数的时间复杂性应为 $\Theta(n)$ 。

\*22. 针对如下情况完成练习21：二叉搜索树中有一些元素的关键值相同。新类的名称为DIndexedAVLtree。

23. 解释一下如何用AVL树将5.5.3节中提到的火车车厢重排问题解决方法的时间复杂性减少到 $O(n \log k)$ 。

\*24. 设计一个类IndexedAVLList，将线性表描述为一棵二叉树，该二叉树与AVL树的区别仅在于它可能不是一棵二叉搜索树。类应支持程序3-1中所定义的所有线性表操作。除了Search函数，其他所有函数的运行时间不应超过对数时间。

## 11.3 红-黑树

### 11.3.1 基本概念

红-黑树 (red-black tree) 是这样的一棵二叉搜索树：树中的每一个节点的颜色是黑色或红色。红-黑树的其他特征可以用相应的扩充二叉树来说明。回忆一下9.4.1节，在一个规则的



二叉树中，用外部节点来替换每一个空指针，就得到了一棵扩充的二叉树。

RB1：根节点和所有外部节点的颜色是黑色。

RB2：根至外部节点途中没有连续两个节点的颜色是红色。

RB3：所有根至外部节点的路径上都有相同数目的黑色节点。

另一种等价的定义源于一个节点与其子女间指针的颜色。从父节点至黑色孩子的指针是黑色的，而至红色孩子的指针是红色的。

RB1'：从内部节点指向外部节点的指针是黑色的。

RB2'：从根至外部节点的途中没有两个连续的红色指针。

RB3'：所有根至外部节点的路径上都有相同数目的黑色指针。

注意，如果知道指针的颜色，就能够推断节点的颜色，反之亦然。在图11-14的红-黑树中，方块是外部节点，粗线是黑色指针，细线是红色指针。可以从指针的颜色和特征 RB1推断出节点的颜色。节点5，50，62和70是红色的，因为它们的父节点指向它们的指针是红色的，其余的节点是黑色的。注意，从根至外部节点的每条路径上都有两个黑色指针和三个黑色节点（包括根和外部节点）；不存在含有两个连续红色节点或指针的路径。

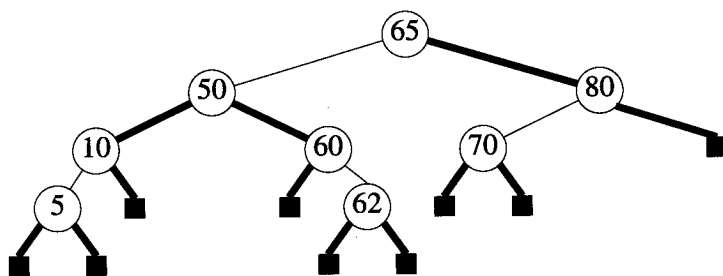


图11-14 红-黑树

设红-黑树中某节点的阶（rank）是从该节点到其子树中任意外部节点的任一条路径上的黑色指针的数量。因此，外部节点的阶是零，图11-14中根节点的阶是2，其左孩子的阶是2，右孩子的阶是1。

定理11-1 设从根到外部节点的路径的长度（length）是该路径中指针的数量。如果  $P$  和  $Q$  是红-黑树中的两条从根至外部节点的路径，那么

$$\text{length}(P) \leq 2\text{length}(Q)$$

证明 考察任意一棵红-黑树。假设根节点的阶是  $r$ ，由RB1' 知每条从根至外部节点的路径中最后一个指针为黑色，从RB2' 知不存在包含连续两个红色指针的路径。因此，每个红色指针的后面都会跟着一个黑色指针，因而，每一条从根至外部节点的路径上都有  $r \sim 2r$  个指针，故有  $\text{length}(P) \leq 2\text{length}(Q)$ 。为了验证上限的可能性，可参考图11-14中的红-黑树。从根至5的左孩子的路径长度是4，而到80的右孩子的长度是2。

定理11-2 设  $h$  是一棵红-黑树的高度（不包括外部节点）， $n$  是树中内部节点的数量，而  $r$  是根节点的阶，则

- 1)  $h \leq 2r$ 。
- 2)  $n \geq 2^r - 1$ 。
- 3)  $h \leq 2\log_2(n+1)$ 。



证明 在定理 11-1 的证明中, 我们知道从根至外部节点的路径的长度不会超过  $2r$ , 因此  $h \leq 2r$  (图 11-14 中除去外部节点的红-黑树的高度是  $2r=4$ )。

因为根节点的阶是  $r$ , 所以从第 1 层至第  $r$  层没有外部节点, 因而在这些层中有  $2^r - 1$  个内部节点。也即内部节点的总数至少应为  $2^r - 1$  (在图 11-14 的红-黑树中, 第 1 和 2 层共有  $2^2 - 1 = 3$  个内部节点, 而在第 3 和 4 层中还包含了其他内部节点)。

由 2) 可以得到  $r \leq \log_2(n+1)$ , 与 1) 合起来即得到 3)。

由于红-黑树的高度最多是  $2\log_2(n+1)$ , 所以, 搜索、插入和删除操作 (在  $O(h)$  时间内完成) 的复杂性为  $O(\log n)$ 。

注意, 最坏情况下的红-黑树的高度大于最坏情况下具有相同 (内部) 节点数目的 AVL 树的高度 (近似于  $1.44\log_2(n+2)$ )。

### 11.3.2 红-黑树的描述

虽然在定义红-黑树时, 将外部节点包括进来非常方便, 但在执行过程中, 我们仍然愿意用零指针或空指针, 而不是物理节点来描述这些节点。进一步的说, 由于指针的颜色与节点的颜色是紧密联系的, 因此对于每个节点, 需要储存的只是该节点的颜色或指向它的两个孩子的指针的颜色。存储每个节点的颜色只需要附加一个位, 而存储每个指针的颜色则需要两位。既然两种方案需要的空间几乎相同, 所以应该基于红-黑树算法的实际运行时间来做出选择。

在插入和删除操作的讨论中, 只对节点颜色的改变做明确的说明, 相应的指针颜色的变化可由推断得到。

### 11.3.3 红-黑树的搜索

可以使用对普通二叉搜索树进行搜索的代码 (见程序 11-2) 来完成对红-黑树的搜索。原代码的复杂性为  $O(h)$ , 对于红-黑树则为  $O(\log n)$ 。由于用相同的代码来搜索普通二叉搜索树、AVL 树和红-黑树, 并且在最坏情况下 AVL 树的高度是最小的, 因此, 在那些以搜索操作为主的应用中, 在最坏情况下 AVL 树能获得最优的时间复杂性。

### 11.3.4 红-黑树的插入

可以利用普通二叉树的插入算法 (见程序 11-3) 将元素插入到红-黑树。当将新元素插入到红-黑树中时, 需要为它上色, 如果插入前是空树, 那么新节点是根节点, 颜色必须是黑色 (参看特征 RB1)。假设插入前树非空, 如果新节点的颜色被赋予黑色, 那么在从根到外部节点的路径中, 将有一个特殊的黑色节点作为新节点的孩子。另一方面, 如果新节点被赋予红色, 那么可能出现两个连续的红色节点。把新节点赋为黑色将肯定导致违反 RB3, 而把新节点赋为红色将可能违反, 也可能不违反 RB2, 因此, 应将新节点赋为红色。

如果将新节点赋为红色而导致违反特征 RB2, 就说树的平衡被破坏了。通过检查新节点  $u$ , 其父节点  $pu$  及祖父节点  $gu$ , 可以确定不平衡的类型。考察违反 RB2 后的情况。现在有两个连续的红色节点, 一个是  $u$ , 另一个一定是它的父节点, 因此  $pu$  存在。因为  $pu$  是红色的, 它不可能是根 (由特征 RB1 知根是黑色的), 那么  $u$  必然有一个祖父节点  $gu$ , 并且它的颜色是黑色的 (特征 RB2)。当  $pu$  是  $gu$  的左孩子,  $u$  是  $pu$  的左孩子且  $gu$  的另一个孩子是黑色时 (这种情况包括  $gu$  的另一个孩子是外部节点的情况), 不平衡是 LLb 类型。其他的不平衡类型是 LLr ( $pu$  是  $gu$  的左孩子,  $u$  是  $pu$  的左孩子且  $gu$  的另一个孩子是红色的), LRb ( $pu$  是  $gu$  的左孩子,  $u$  是  $pu$  的

右孩子且  $gu$  的另一个孩子是黑色的), LLb, LRr, RRb, RRr, RLb 和 RLr。

XYr (X 和 Y 既可以是 L, 也可以是 R) 类型的不平衡可以通过改变颜色来处理, 而 XYb 类型则需要使用旋转。当改变一个节点的颜色时, 树中可能有两层违反 RB2。这时需要对新层进行重新分类, 将  $u$  变为  $gu$ , 然后再次进行转换。旋转结束后, 不再违反 RB2, 因此不需要再进行其他操作。

图11-15给出了LLr 和LRr 型不平衡的颜色变化, 这些颜色的变化是一致的。黑色节点用深色阴影表示, 而红色节点用浅色阴影表示, 例如, 在图 11-15中,  $gu$  是黑色节点,  $u$  和  $pu$  是红色节点。从  $gu$  到它的左右孩子的指针是红色的,  $gu_R$  是  $gu$  的右子树,  $pu_R$  是  $pu$  的右子树, LLr 和LRr 颜色的改变都需要我们将  $pu$  的颜色和  $gu$  右孩子的颜色由红色改为黑色。另外, 如果  $gu$  不是根, 还要将  $gu$  的颜色由黑色改为红色。由于  $gu$  是根节点时其颜色没有改变, 因此, 若  $gu$  是红-黑树的根节点, 所有从根至外部节点路径上的黑色节点的数量都增加了一。

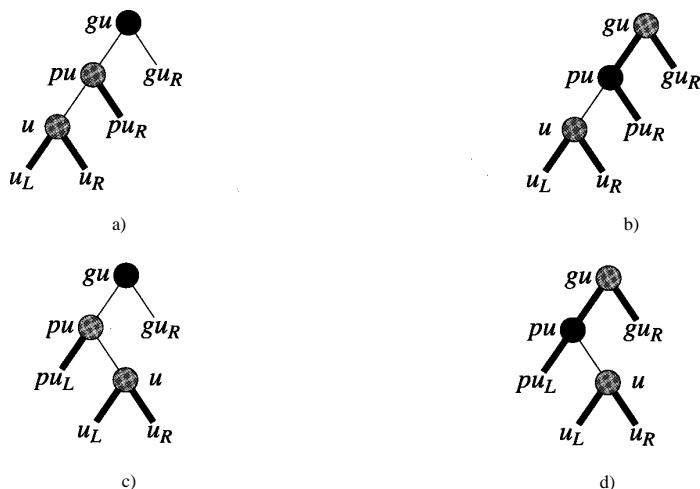


图11-15 LLr 和LRr 类型的颜色变化

a) LLr 型不平衡 b) LLr 颜色改变之后 c) LRr 型不平衡 d) LRr 颜色改变之后

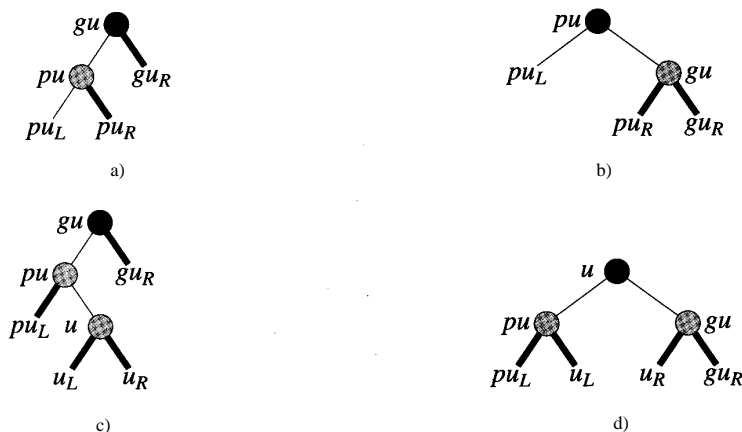


图11-16 红-黑树插入操作后的LLb和LRb旋转

a) LLb 型不平衡 b) LLb 旋转之后 c) LRb 型不平衡 d) LRb 旋转之后

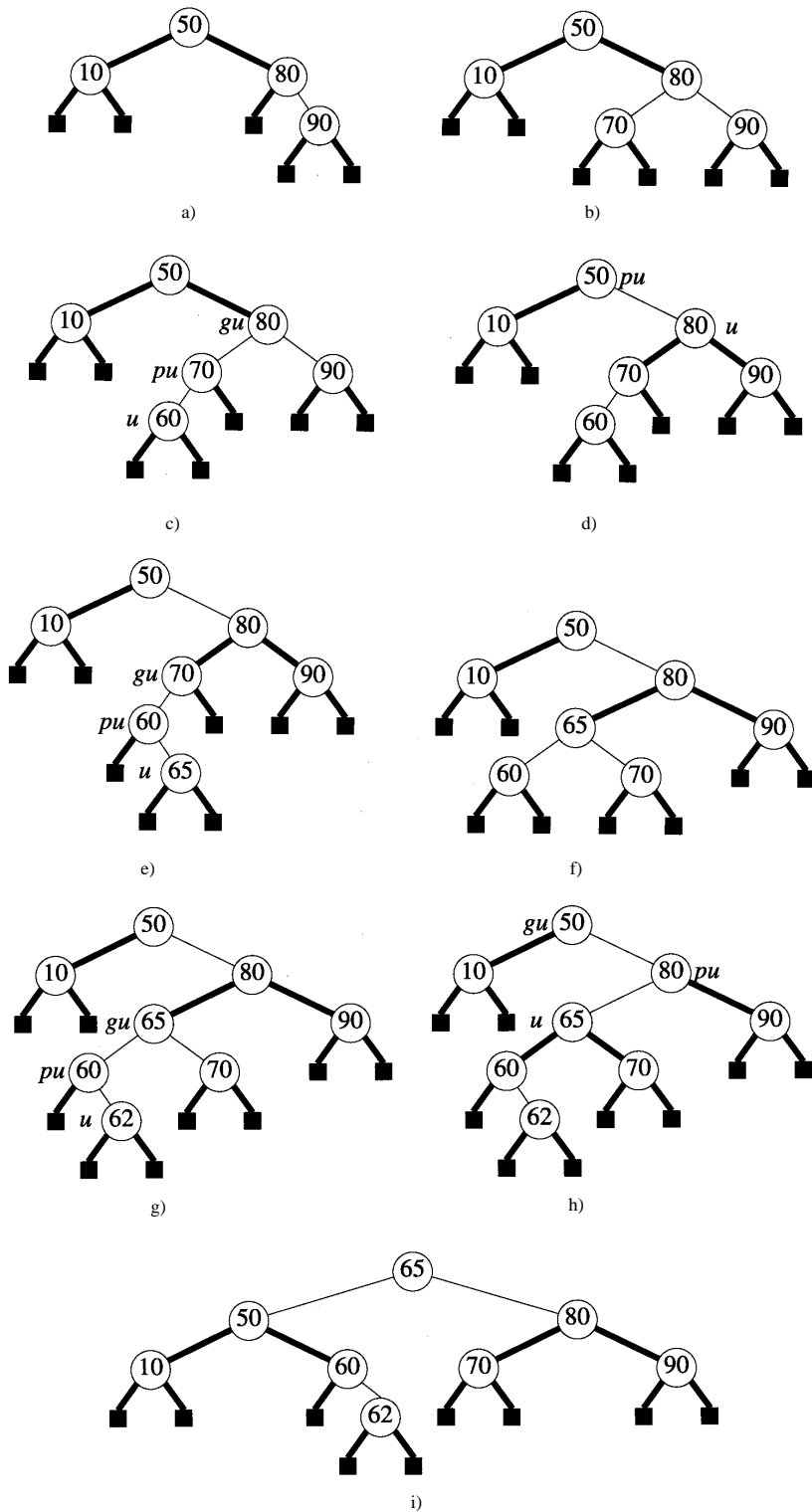


图11-17 红-黑树的插入

a) 初始状态 b) 插入70 c) 插入60 d) LLr 颜色 e) 插入65 f) LRb 旋转 g) 插入62 h) LRR 颜色 i) RLb 旋转

如果将  $gu$  的颜色改为红色而引起了不平衡,那么  $gu$  就变成了新的  $u$  节点,它的双亲就变成了新的  $pu$ ,它的祖父节点就变成了新的  $gu$ ,需要继续恢复平衡。如果  $gu$  是根节点或者  $gu$  节点的颜色改变没有违反规则RB2,那么工作就完成了。

图11-16给出了处理LLb和LRb不平衡的旋转操作。在图11-16a和b中, $pu$ 是 $pu_L$ 的根。注意这些旋转与AVL树的插入操作完成后,用来处理不平衡的LL(如图11-8所示)和LR(如图11-9所示)旋转之间的相似之处。指针的改变是相同的,例如,在LL和LR旋转中,由于指针改变了,故需要将 $gu$ 的颜色由黑色改为红色,而 $pu$ 的颜色由红色改为黑色。

检查图11-16中旋转之后的节点(或指针)颜色,会发现在所有从根至外部节点的路径上,黑色节点(指针)的数量是不变的,进一步说,相关子树的根(旋转前的 $gu$ 和旋转后的 $pu$ ) 在旋转后是黑色的,因此,在从根节点至新的 $pu$ 的路径上,连续两个红色节点是不存在的,不需要再作平衡。插入后,一次旋转( $O(\log n)$ 次颜色改变)已经足够保持平衡了。

例11-1 考察图11-17a所示的红-黑树,该图只给出了指针颜色,节点颜色可以从指针颜色以及根节点一般是黑色等常识中推断。为了方便理解,给出了外部节点。实际上,指向外部节点的指针只是简单的空和零,并且外部节点不必专门描述。注意,所有从根至外部节点的路径中都有两个黑色指针。

现在,采用程序11-3的算法,将70插入到红-黑树中。新节点作为80的左孩子插入到树中,由于插入在一棵非空的树中进行,新节点被赋予红色,因此,从父节点(80)指向它的指针也是红色。这次插入操作不会导致违反RB2,所以不需要矫正。

接下来,把60插入到图11-17b的树中,程序11-3的算法将把新节点作为70的左孩子,如图11-17c。新节点为红色,指向它的指针也为红色。新节点是 $u$ 节点,其父节点(70)是 $pu$ ,祖父节点(80)是 $gu$ ,由于 $pu$ 和 $u$ 都是红色,这里就存在一个不平衡,这个不平衡是LLr类型的不平衡( $pu$ 是 $gu$ 的左孩子, $u$ 是 $pu$ 的左孩子, $gu$ 的另一个孩子是红色)。执行图11-15a和b的颜色改变,得到图11-17d。现在, $u$ , $pu$ 和 $gu$ 节点都上升了两层,节点80是新的 $u$ 节点,根节点是 $pu$ , $gu$ 是零。由于没有 $gu$ 节点,所以这里不会产生违反RB2类型的不平衡,所有从根到外部节点的路径实际上都有两个黑色指针。

现在将65插入到图11-17d的树中,结果如图11-17e所示。新节点是 $u$ 节点,它的父节点和祖父节点分别是 $pu$ 和 $gu$ 节点。这里产生了一个LRb类型的不平衡,需要执行图11-16c和d的旋转,结果如图11-17f所示。

最后,将62插入到树中,得到图11-17g,产生一个LRr类型的不平衡,需要进行颜色的改变。图11-17h给出了所得到的树和新的 $u$ , $pu$ 和 $gu$ 节点。颜色的改变引起了RLb类型的不平衡,必需执行RLb旋转,旋转后的结果如图11-17i所示。旋转后,不需要再作任何工作,红-黑树的插入操作完成。

### 11.3.5 红-黑树的删除

对于删除操作,首先使用普通二叉搜索树的删除算法(程序11-4),然后进行颜色的矫正,如果需要的话,还要作一次单旋转。考察图11-18a中的红-黑树,如果用程序11-4删除70,将得到图11-18b所示的树(如果给出了指针颜色,还需要改变90的左指针的颜色)。当从a中删除90时,得到树c(如果使用了指针颜色,那么65的右指针的颜色也需要改变)。从树a中删除65得到树d(重复强调一次,如果使用了指针颜色,那么指针颜色也要改变)。设 $y$ 是替代被删

除节点的节点,如图11-18所示。在图11-18b的情况中,90的左孩子被删除了,它的新的左孩子是外部节点 $y$ 。

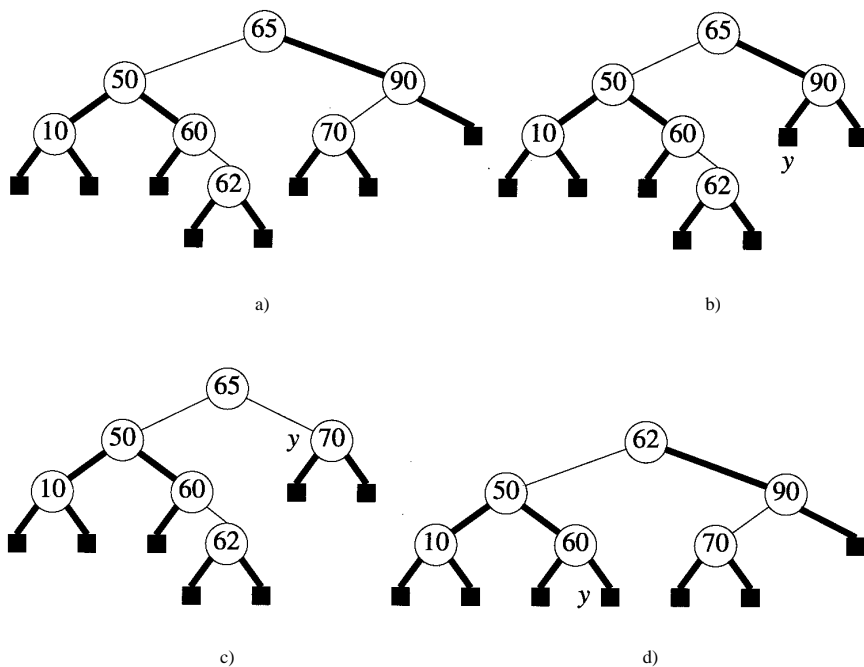


图11-18 红-黑树的删除

a) 初始状态 b) 删除70 c) 删除90 d) 删除65

在树b的情况中,被删除节点(树a中的70)是红色的,删除它不会影响从根至外部节点的路径中黑色节点的数量,因此不需要作任何矫正工作。在树c中,被删除节点(树a中的90)是黑色的,那么,从根至外部节点路径中黑色节点(和指针)的数量比删除前少了一个。由于 $y$ 不是新的根,因此违反了RB3,而在树d中被删除节点是红色的,所以不会出现违反RB3的情况。仅当被删除节点是黑色的且 $y$ 不是所得树的根时,会出现违反RB3的情况。用程序11-4执行删除操作后不可能出现违反其他红-黑树特征的情况。

当违反RB3的情况发生时,以 $y$ 为根节点的子树缺少一个黑色节点(或一个黑色指针),因此,在从根至 $y$ 子树的外部节点的路径上的黑色节点数量比从根至其他外部节点路径上的黑色节点数量少一个,这时树是不平衡的。通过识别 $y$ 的父节点 $p_y$ 和同胞节点 $v$ 来区分不平衡的性质,当 $y$ 是 $p_y$ 的右孩子时,不平衡是R类型的,否则不平衡是L类型的。可以看到,如果 $y$ 缺少一个黑色节点,那么 $v$ 就肯定不是一个外部节点。如果 $v$ 是一个黑色节点,那么不平衡是Lb或Rb类型的;而当 $v$ 是红色节点时,不平衡是Lr或Rr类型的。

首先考察Rb类型的不平衡。Lb型不平衡的处理与之相似。根据 $v$ 的红色子女的数量,把Rb型不平衡分为三种子情况:Rb0, Rb1和Rb2。

当不平衡类型是Rb0时,需要执行颜色的改变(如图11-19所示)。图11-19给出了 $p_y$ 颜色的两种可能改变。如果 $p_y$ 是黑色的,那么颜色的改变将导致以 $p_y$ 为根的子树缺少一个黑色节点,并且,在图11-19b中,从根至 $v$ 的外部节点路径上的黑色节点数量也减少了一个,因此,颜色改变后,无论路径是到 $v$ 的外部节点还是到 $y$ 中的外部节点都会缺少一个黑色节点。如果

$py$  是整棵红-黑树的根，那么就不需要再作其他工作，否则， $py$  就成为新的  $y$ ， $y$  的不平衡需要重新划分，并且在这个新的  $y$  点再进行合适的矫正工作。

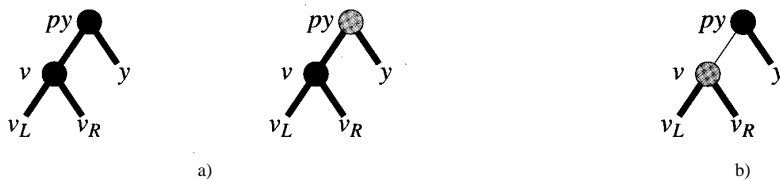


图11-19 红-黑树删除操作的Rb0 颜色改变

a) Rb0 型不平衡 b) Rb0 颜色改变

若改变颜色前  $py$  为红色，则从  $y$  到外部节点路径上的黑色节点数量增加了一个，而  $v$  中的没有改变，整棵树就达到了平衡。

不平衡类型是 Rb1 和 Rb2 时，需要进行旋转，如图 11-20 所示。图中的非阴影节点表示那些既可能是红色，也可能是黑色的节点。这种节点的颜色在旋转后不会发生变化，因此，图 11-20b 中所示子树的根在旋转前和旋转后，颜色保持不变。b 中  $v$  的颜色与 a 中  $py$  的颜色是一样的。你应当能够发现在旋转后， $y$  中从根至外部节点路径上的黑色节点（指针）数量增加了一个，而从根至其他外部节点路径上的黑色节点的数量没有变化，旋转使树恢复了平衡。

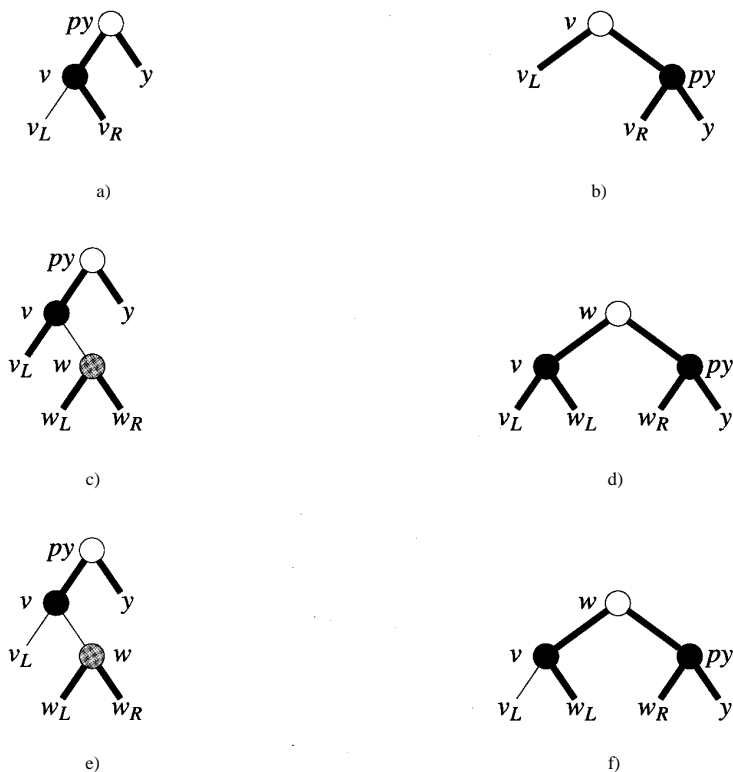


图11-20 红-黑树删除操作中Rb1 和Rb2 型不平衡的旋转

a) Rb(i) 型不平衡 b) Rb(i) 旋转之后 c) Rb1(ii) 型不平衡 d) Rb1(ii) 旋转之后 e) Rb2 不平衡 f) Rb2 旋转之后

接下来考察Rr 类型的不平衡， $Lr$  型不平衡与它是对称的。由于 $y$  中缺少一个黑色节点并且 $v$  是红色的， $v_L$  和 $v_R$  中至少有一个黑色节点不是外部节点，因此， $v$  的孩子都是内部节点。根据 $v$  的右孩子中红色孩子（0，1或2）的数量，可以进一步把Rr 类型的不平衡划分为三种情况，这三种情况都可以用旋转来处理。如图11-21和11-22所示，可以验证其中的旋转使整棵树恢复了平衡。



图11-21 红-黑树删除操作的Rr0 旋转

a) Rr0 型不平衡 b) Rr0 旋转

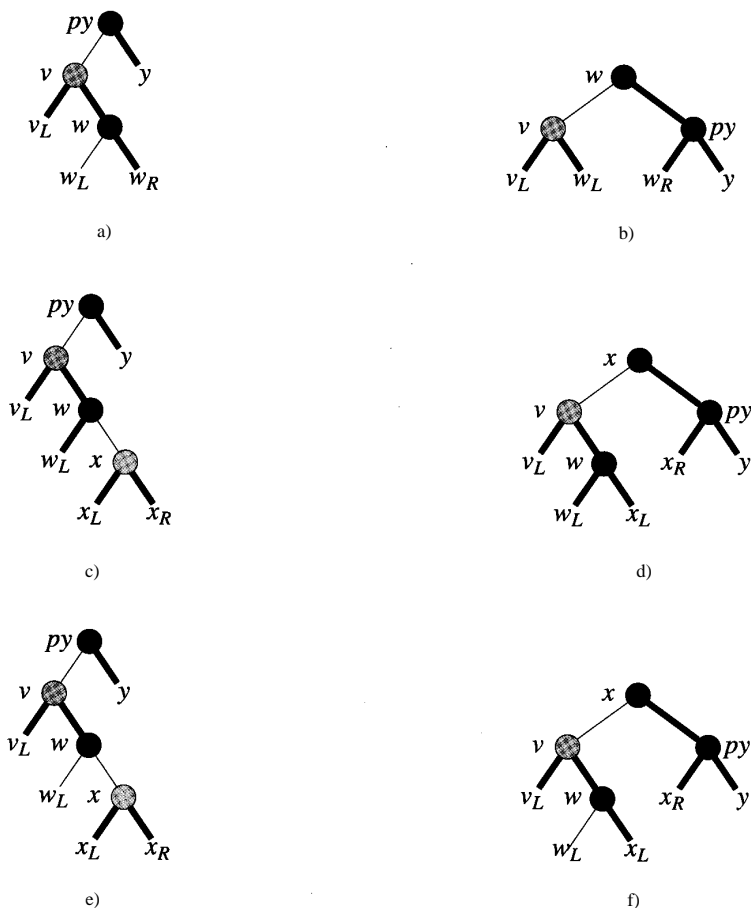


图11-22 红-黑树删除操作的Rr1 和Rr2 旋转

a) Rr1(i) 型不平衡 b) Rr1(i) 旋转之后 c) Rr1(ii) 型不平衡 d) Rr1(ii) 旋转之后 e) Rr2 型不平衡 f) Rr2 旋转之后

例11-2 如果从图11-17i 的红-黑树中删除90，就得到图11-23所示的树。由于被删除节点不是



根节点且是黑色的，因此产生了Rb0型不平衡。执行颜色的改变后得到了图11-23b中的树。由于py原来是红色的，因此改变颜色后使树重新恢复了平衡。

如果现在从树b中将80删除，就得到了树c。由于删除的是红色节点，删除后树仍然是平衡的。从树c中将70删除，得到树d，这次删除的是非根黑色节点，树的平衡被破坏了，不平衡类型是Rr1(ii)类型( $v$ 的右孩子 $w$ 的右孩子指针是红色的)。执行Rr1(ii)型旋转后得到树e，树的平衡得到了恢复。

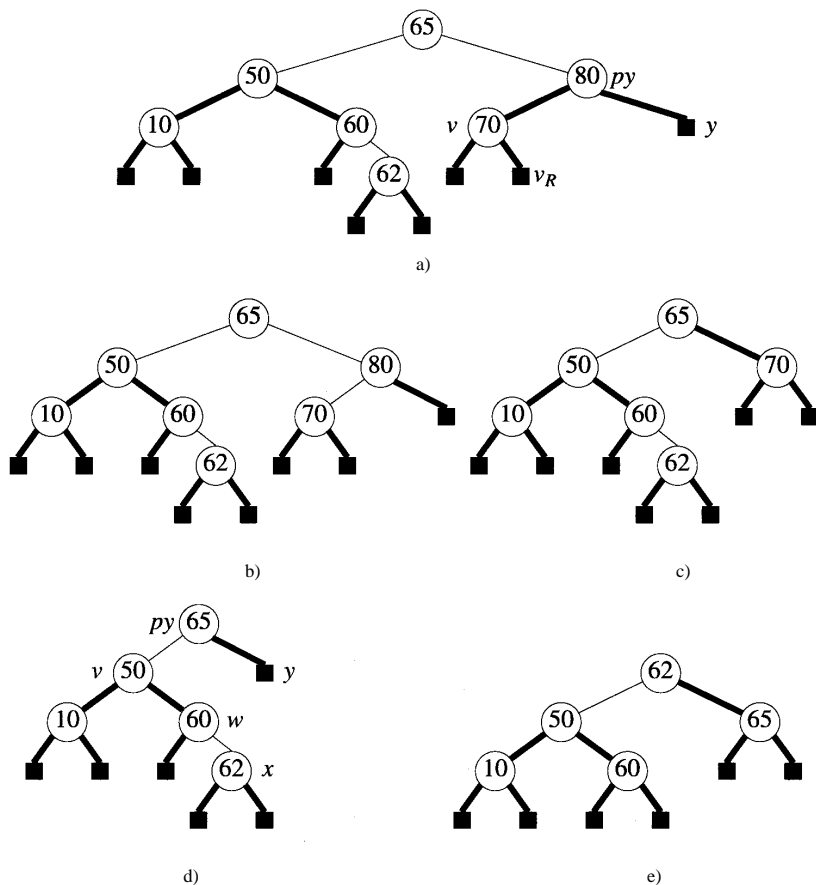


图11-23 红-黑树的删除

a) 删除90 b) Rb0 颜色改变之后 c) 删除80 d) 删除70 e) Rr1(ii) 旋转之后

### 11.3.6 实现细节的考虑及复杂性分析

在插入或删除操作后，为了重新恢复红-黑树的平衡而采取的矫正方法需要我们回到从根节点至插入或删除节点的路径上来。如果除数据、左孩子、右孩子和颜色域外每个节点还有一个双亲域，那么这种回溯很容易实现。还有一种方法是将从根节点至插入/删除节点路径上所遇到的每个节点的指针保存到一个堆栈内，通过从堆栈中删除这些指针，就可以返回到根节点。对于一个 $n$ 元素的红-黑树，增加双亲域使得对空间的需求增加了 $\Theta(n)$ ，而使用堆栈则使空间的需求增加了 $\Theta(\log n)$ 。虽然堆栈方法在空间上更节省，但双亲指针方法运行得更快一些。

插入或删除后,由于颜色的改变是沿着向根节点的方向进行的,故需要时间  $O(\log n)$ 。另一方面,旋转能够保证树的重新平衡。每次插入/删除操作最多需要一次旋转。每次颜色改变或旋转操作需要的时间是  $\Theta(1)$ ,因此插入/删除操作需要的总时间是  $O(\log n)$ 。

## 练习

25. 画出对应于图 11-15 中 LLr 和 LRr 类型的 RRr 和 RLr 类型的颜色改变。
26. 画出对应于图 11-16 中 LLb 和 LRb 类型的 RRb 和 RLb 类型的颜色改变。
27. 画出对应于图 11-19 中 Rb0 类型的 Lb0 类型的颜色改变。
28. 画出 Lb1 和 Lb2 类型的旋转示意图,相对应的 Rb1 和 Lb2 类型的旋转如图 11-20 所示。
29. 画出 Lr0、Lr1 和 Lr2 类型的旋转示意图,相对应的 Rr0、Rr1 和 Rr2 类型的旋转如图 11-21 和图 11-22 所示。

\*30. 设计一个 C++ 类 RedBlackTree, 它包括二叉搜索树的函数 Search、Insert、Delete 和 Ascend。编写所有函数并检验其正确性。证明前三种操作的复杂性是  $O(\log n)$ , 最后一种操作的复杂性是  $\Theta(1)$ 。Insert 和 Delete 函数的实现必须采用本节所讨论的方法。

## 11.4 B-树

### 11.4.1 索引顺序访问方法

当字典足够小,可以驻留在内存中时,AVL 树和红-黑树都能够保证获得很好的性能。对于较大的字典(外部字典或文件),它们必须存储在磁盘上,可以通过采用度数高的搜索树来改善字典操作的性能。在研究高度数搜索树之前,先看一下用于外部字典的索引顺序访问方法(indexed sequential access method, ISAM),这种方法提供了很好的顺序和随机访问。

在 ISAM 方法中,可用的磁盘空间被划分为很多块,块是磁盘空间的最小单位,被用来作为输入和输出。块一般具有与磁道同样的长度,且可以用单个搜索和很小的延迟进行输入输出。字典元素以升序存储在块中。

在顺序访问时,依次输入各个块,在每个块中按升序搜索元素。如果每个块包含  $m$  个元素,则搜索每个元素所需要的磁盘访问次数为  $1/m$ 。

要支持随机访问,索引是不可缺少的。索引中包括每个块中的最大关键值。由于索引中所包含的关键值数量仅与块数相同,并且每个块一般都能贮存很多元素( $m$  值通常较大),因此索引足以驻留在内存中。对关键值为  $k$  的元素作一次随机访问,首先只要寻找包含相应元素的块的索引,然后将相应的块从磁盘中取出并在其中寻找需要的元素。这样,执行一次随机访问只需要一次磁盘访问就足够了。

这种技术可以扩充到更大的字典,这种字典能跨越几个磁盘。现在,元素按升序被分配到各个磁盘以及每个磁盘的不同块中。每个磁盘都有一个块索引,其中保留了该磁盘每个块中的最大关键值。另外,有一个磁盘索引保存每个磁盘中的最大关键值,这个索引一般驻留在内存中。

在上述情况下如果要执行一个随机访问,首先需在磁盘索引中进行搜索以判断所需要的记录可能存储在哪个磁盘上,找到磁盘后,取出相应磁盘的块索引,并在其中搜索所需要的块,块被取出后,在其内部搜索所需要的记录。

由于 ISAM 方法本质上是一种公式化描述方法,当执行插入和删除时它就会陷入困境。通

过在每个块中留一些空间可以部分减轻这种困难，这样在执行少量的插入时可以不必要在块之间移动元素。类似地，在删除操作后可把空间保留下来，以避免在块之间进行代价昂贵的元素移动。

### 11.4.2 m 叉搜索树

定义 [m 叉搜索树]  $m$  叉搜索树 ( $m$ -way search tree) 可以是一棵空树，如果非空，它必须满足以下特征：

- 1) 在相应的扩充搜索树中 (用外部节点替换零指针)，每个内部节点最多可以有  $m$  个子节点及  $1 \sim m-1$  个元素 (外部节点不含元素和子女)。
- 2) 每个含  $p$  个元素的节点，有  $p+1$  个子节点。
- 3) 考察含  $p$  个元素的任意节点。设  $k_1, \dots, k_p$  是这些元素的关键值。这些元素顺序排列，即有  $k_1 < k_2 < \dots < k_p$ 。设  $c_0, c_1, \dots, c_p$  是节点的  $p+1$  个孩子。以  $c_0$  为根的子树中的元素关键值小于  $k_1$ ，而以  $c_p$  为根的子树中的元素关键值大于  $k_p$ ，并且以  $c_i$  为根的子树中的元素关键值会大于  $k_i$  而小于  $k_{i+1}$ ，其中  $1 \leq i \leq p$ 。

虽然在定义  $m$  叉搜索树时把外部节点包括进来很有用，但在实际实现中不需要专门描述外部节点，只需用零指针或空指针来表示外部节点。

图11-24给出了一棵七叉搜索树，图中的黑色方块代表外部节点，所有其他节点都是内部节点。根包含两个元素 (关键值是10和80) 和三个子女。中间的子女有6个元素和7个孩子，其中6个孩子是外部节点。

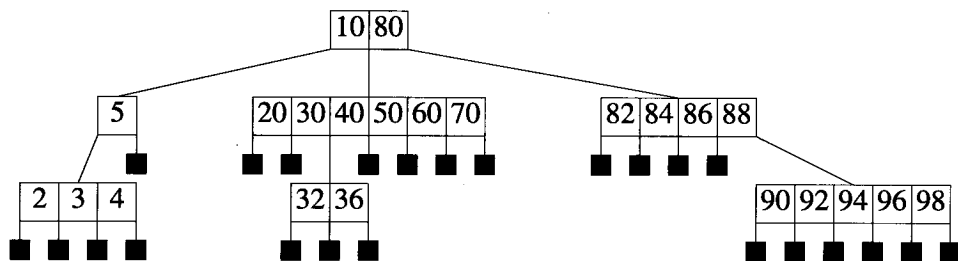


图11-24 七叉树

#### 1. 在 $m$ 叉搜索树中进行搜索

要从图11-24中的七叉搜索树中将关键值为31的元素搜索出来，可先从根节点开始。31位于10和80之间，沿中间的指针往下找。(由定义知，第一棵子树中所有元素的关键值  $< 10$ ，而第三棵子树中所有元素的关键值  $> 80$ )。中间子树的根被搜索。由于  $k_2 < 31 < k_3$ ，故移到该节点的第三棵子树中。此时可以发现  $31 < k_1$ ，所以移动到第一棵子树中。这次移动使我们从树中“掉”了下来，即到达了外部节点。因此可以得出结论，搜索树中不包含关键值为31的元素。

#### 2. 向 $m$ 叉搜索树中插入元素

如果希望将关键值为31的元素插入到树中，则先要按以上步骤搜索31，在节点[32, 36]处搜索失败。由于该节点可以容纳六个元素 (七叉搜索树的每个节点最多可以容纳六个元素)，新元素可以插到该节点的第一个位置。

假定要将65插到树中,则先对65进行搜索,在节点[20,30,40,50,60,70]的第六棵子树中搜索失败,由于此节点不能再容纳额外的元素,所以必须产生一个新节点。新元素放入新节点中,新节点成为节点[20,30,40,50,60,70]的第6个孩子。

### 3. 从 $m$ 叉搜索树中删除元素

从图11-24中的搜索树中删除关键值为20的元素,首先要进行搜索,该元素是根节点中间孩子的第一个元素。由于 $k_1 = 20$ 并且 $c_0 = c_1 = 0$ ,故可以简单地将它从节点中删除,根节点新的中间子女变成[30,40,50,60,70]。类似地,如要删除关键值为84的元素,首先对它定位,它是根节点第三个子女的第二个元素,由于 $c_1 = c_2 = 0$ ,故可以从节点中直接删除该元素,原节点变为[82, 86, 88]。

当删除关键值为5的元素时,需要多作一些工作。由于被删除元素是节点中的第一个元素并且它的相邻子女(这里是 $c_0$ 和 $c_1$ )中至少有1个是非零的,因此,需要从非空相邻子树中找一个元素来替换被删除元素。从左子树( $c_0$ )中,可以将具有最大关键值的元素移上来(这里是关键值为4的元素)

从图11-24中的根节点中删除关键值为10的元素时,既可以用 $c_0$ 中的最大元素,也可以用 $c_1$ 中的最小元素进行替换。如果用 $c_0$ 中的最大元素进行替换,那么元素5将被移上来,而且要对它在原节点中的位置作一次替换,元素4就被移到元素5原来的位置。

### 4. $m$ 叉搜索树的高度

一棵高度为 $h$ 的 $m$ 叉搜索树最少可以有 $h$ 个元素(每层一个节点,每个节点含一个元素),最多可以有 $m^h - 1$ 个元素。此上限取自这种情况:从1到 $h-1$ 层的每个节点都含有 $m$ 个孩子并且第 $h$ 层的节点没有孩子。这样一棵树共有 $\sum_{i=0}^{h-1} m^i = (m^h - 1)/(m - 1)$ 个节点。由于每个节点可以有 $m-1$ 元素,所以元素的总数为 $m^h - 1$ 。

由于高度为 $h$ 的 $m$ 叉搜索树中元素的个数在 $h$ 到 $m^h - 1$ 之间,所以一棵 $n$ 元素的 $m$ 叉搜索树的高度在 $\log_m(n+1)$ 到 $n$ 之间。

例如,一棵高度为5的200叉搜索树能够容纳 $32 \times 10^{10} - 1$ 个元素,但也可以只容纳五个元素。同样,一棵含有 $32 \times 10^{10} - 1$ 个元素的200叉搜索树的高度可以是5,也可以是 $32 \times 10^{10} - 1$ 。当搜索树存储在磁盘上时,搜索、插入和删除时间取决于磁盘的访问次数(假设每个节点不大于一个磁盘块)。由于搜索、插入和删除操作需要的磁盘访问次数是 $O(h)$ ,其中 $h$ 是树的高度,因此,必须确保高度值接近于 $\log_m(n+1)$ 。这种保证可由 $m$ 叉平衡搜索树提供。

## 11.4.3 $m$ 序B-树

定义 [m序B-树]  $m$ 序B-树(B-Tree of order  $m$ )是一棵 $m$ 叉搜索树,如果B-树非空,那么相应的扩充树满足下列特征:

- 1) 根节点至少有2个孩子。
- 2) 除了根节点以外,所有内部节点至少有 $\lceil m/2 \rceil$ 个孩子。
- 3) 所有外部节点位于同一层上。

图11-24中的七叉搜索树不是一棵七序B-树,因为它的外部节点不在同一层上。即使它所有的外部节点在同一层上,它也不会是一棵七序B-树,因为它的非根内部节点[5]有2个孩子,内部节点[32, 36]有3个孩子,而七序B-树的非根节点必须至少有 $\lceil 7/2 \rceil = 4$ 个孩子。图11-25是一棵七序B-树,所有外部节点均位于第三层,根节点有3个孩子,且剩下的所有内部节点至少有4个孩子,此外,它也是一个七叉搜索树。

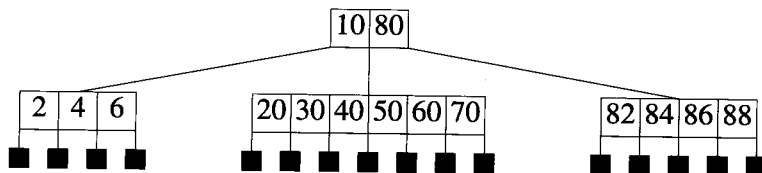


图11-25 七序B-树

在一棵二序B-树中，没有哪个内部节点会有2个以上的孩子。由于在二序B-树中每个内部节点必须至少有2个子女，所以一棵二序B-树的所有内部节点都恰好有2个孩子。这一点以及要求所有外部节点必须在同一层上暗示了二序B-树是一棵满二叉树。正因如此，这些树存在的条件是元素的个数为 $2^h - 1$ ，其中 $h$ 是一个整数。

在一棵三序B-树中，内部节点既可以有2个也可以有3个孩子，因此也把三序B树称作2-3树。由于四序B-树的内部节点必须有2个、3个或4个孩子，这种树也叫作2-3-4树（或简称2,4树）。图11-26中给出了一棵2-3树的例子，虽然该树中没有含4个孩子的内部节点，但只要把关键值为14和16的元素加入到20的左孩子中，就可以得到一棵至少有一个含4个孩子的节点的2-3-4树了。

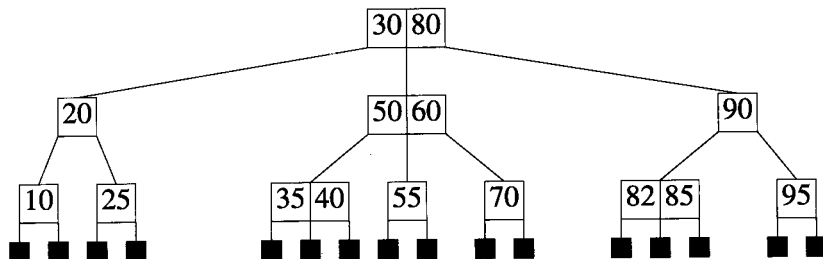


图11-26 2-3树或三序B-树

#### 11.4.4 B-树的高度

定理11-3 设 $T$ 是一棵高度为 $h$ 的 $m$ 序B-树， $d=\lfloor m/2 \rfloor$ 且 $n$ 是 $T$ 中的元素个数，则

$$1) 2d^{h-1} - 1 \leq n \leq m^h - 1.$$

$$2) \log_m(n+1) \leq h \leq \log_d\left(\frac{n+1}{2}\right) + 1.$$

证明  $n$  的上限源于 $T$ 是一棵 $m$ 叉搜索树，这在前面已经证明过了。对于下限，注意相应的扩充B-树的外部节点都在 $h+1$ 层，而1, 2, 3, 4, ...,  $h+1$ 层的节点最小数目是1, 2,  $2d$ ,  $2d^2$ , ...,  $2d^{h-1}$ ，因此B-树中外部的最小数是 $2d^{h-1}$ 。由于外部节点的数量比元素的个数多1，因此

$$n \geq 2d^{h-1} - 1$$

从1)直接可以得到2)

由定理11-3可知，一棵高度为3的200序B-树中至少有19 999个元素，而高度为5的200序B-树中至少有 $2 \times 10^8 - 1$ 个元素。因此，如果使用200序或更高序B-树，即使元素数量再多，树的高度也可以很小。实际上，B-树的序取决于磁盘块的大小和单个元素的大小。节点小于磁盘块的大小并无好处，这是因为每次磁盘访问只读或写一个块。节点大于磁盘块的大小会带

来多重磁盘访问，每次磁盘访问都伴随一次搜索和时间延迟，因此节点大于磁盘块的大小也是不可取的。

虽然在实际应用中，B-树的序很大，但在我们的例子中所用的 $m$ 值很小，因为一棵两层的 $m$ 序B-树至少有 $2d-1$ 个元素，当 $m$ 的值为200， $d$ 为100时，一棵两层的200序B-树至少有199个元素，处理一棵如此多元素的树非常麻烦。

#### 11.4.5 B-树的搜索

B-树的搜索算法与 $m$ 叉搜索树的搜索算法相同。在搜索过程中，从根至外部节点路径上的所有内部节点都有可能被搜索到，因此，磁盘访问次数最多是 $h$ （ $h$ 是B-树的高度）。

#### 11.4.6 B-树的插入

将一个元素插入B-树中时，首先要检查具有相同关键值的元素是否存在，如果找到了这个元素，那么插入失败，因为不允许重复值存在。当搜索不成功时，便可以将元素插入到搜索路径中所遇到的最后一个内部节点处。例如，当将关键值为3的元素插入到图11-25的B-树中时，首先检查根节点及其左孩子，在左孩子的第二个外部节点处搜索失败。由于左孩子可以容纳六个元素而目前只有三个，因此新元素可以直接插入到这个节点中。结果如图11-27a所示。对根节点及左孩子有两次磁盘读操作，而对左孩子另有一次磁盘写操作。

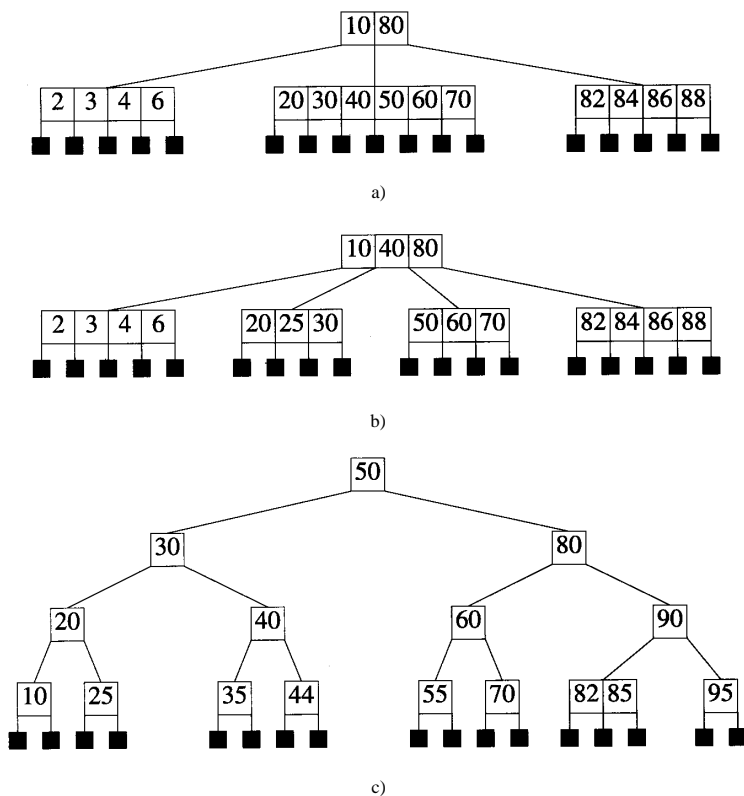


图11-27 B-树的插入

a) 向图11-25中插入30 b) 向a) 中插入25 c) 向图11-26中插入44



下面，将关键值为 25 的元素插入到图 11-27a 的 B-树中，这个元素将被插入到节点 [20,30,40,50,60,70] 中，但该节点已经饱和了。当新元素需要插入到饱和节点中时，饱和节点需要被分开。设  $P$  是饱和节点，现将带有空指针的新元素  $e$  插入到  $P$  中，得到一个有  $m$  个元素和  $m+1$  个孩子的溢出节点。用下面的序列表示溢出节点：

$$m, c_0, (e_1, c_1), \dots, (e_m, c_m)$$

其中  $e_i$  是元素， $c_i$  是孩子指针。从  $e_d$  处分开此节点，其中  $d = \lceil m/2 \rceil$ 。左边的元素保留在  $P$  中，右边的元素移到新节点  $Q$  中， $(e_d, Q)$  被插入到  $P$  的父节点中。新的  $P$  和  $Q$  的格式为：

$$P: d-1, c_0, (e_1, c_1), \dots, (e_{d-1}, c_{d-1})$$

$$Q: m-d, c_d, (e_{d+1}, c_{d+1}), \dots, (e_m, c_m)$$

注意  $P$  和  $Q$  的孩子数量至少是  $d$ 。

在本例中，溢出节点是

$$7, 0, (20,0), (25,0), (30,0), (40,0), (50,0), (60,0), (70,0)$$

且  $d=4$ 。从  $e_4$  处分开后的两个节点是：

$$P: 3, 0, (20,0), (25,0), (30,0)$$

$$Q: 3, 0, (50,0), (60,0), (70,0)$$

当把  $(40,Q)$  插入到  $P$  的父节点中时，得到图 11-27b 所示的 B-树。

将 25 插入到图 11-27a，需要从磁盘中得到根节点及其中间孩子，然后将分开的两个节点和修改后的根节点写回到磁盘中，磁盘访问次数一共是 5 次。

来看最后一个例子。考察将关键值为 44 的元素插入到图 11-26 的 2-3 树中，此元素将插入到节点 [35,40] 中，由于该节点是饱和节点，故得到溢出节点：

$$3, 0, (35,0), (40,0), (44,0)$$

从  $e_d = e_2$  处分开得到 2 个节点：

$$P: 1, 0, (35,0)$$

$$Q: 1, 0, (44,0)$$

当把  $(40,Q)$  插入到  $P$  的父节点  $A$  中时，发现该节点也是饱和的，插入后，又得到溢出节点：

$$A: 3, P, (40,Q), (50,C), (60,D)$$

其中  $C$  和  $D$  是指向节点 [55] 和 [70] 的指针。溢出节点  $A$  被分开，产生节点  $B$ 。新节点  $A$  和  $B$  如下：

$$A: 1, P, (40,Q)$$

$$B: 1, C, (60,D)$$

现在需要将  $(50,B)$  插入到根节点中，在此之前根节点的结构是：

$$R: 2, S, (30,A), (80,T)$$

其中  $S$  和  $T$  是分别指向根节点第一和第三棵子树的指针。插入完成后，得到溢出节点：

$$R: 3, S, (30,A), (50,B), (80,T)$$

将此节点从关键值为 50 的元素处分开，并产生一个新节点  $R$  和一个新节点  $U$ ，如下所示：

$$R: 1, S, (30,A)$$

$$U: 1, B, (80,T)$$

$(50,U)$  一般应插入到  $R$  的父节点中，但是  $R$  没有父节点，因此，产生一个新的根节点如下：

$$1, R, (50,U)$$

得到的 2-3 树如图 11-27c 所示。

读取节点 [30,80]、[50,60] 和 [35,40] 时执行了 3 次磁盘访问。对每次节点分裂，将修改的节



点和新产生的节点写回磁盘需执行 2 次磁盘访问, 由于有 3 个节点被分开, 因此需执行 6 次写操作。最后产生一个新的根节点并写回磁盘, 又需占用 1 次额外的磁盘访问, 因此磁盘访问的总次数为 10。

当插入操作引起了  $s$  个节点的分裂时, 磁盘访问的次数为  $h$  (读取搜索路径上的节点) +  $2s$  (回写两个分裂出的新节点) + 1 (回写新的根节点或插入后没有导致分裂的节点)。因此, 所需要的磁盘访问次数是  $h+2s+1$ , 最多可达到  $3h+1$ 。

#### 11.4.7 B-树的删除

删除分为两种情况: 1) 被删除元素位于其孩子均为外部节点的节点中 (即元素在树叶中); 2) 被删除元素在非树叶节点中。既可以用左相邻子树中的最大元素, 也可以用右相邻子树中的最小元素来替换被删除元素, 这样 2) 就转化为 1)。替换元素必须确保在树叶中。

考察从图 11-27a 的 B-树中删除关键值为 80 的元素。由于元素不在树叶中, 需要找一个合适的替换。关键值为 70 (左相邻子树中的最大元素) 和 82 (右相邻子树中的最小元素) 成为候选对象。当选用 70 时, 还存在将此元素从树叶中删除的问题。

如果要从图 11-27c 的 2-3 树中将关键值为 80 的元素删除, 既可以用 70, 也可以用 82 来替换此元素。如果选择 82, 那么还有从树叶 [82, 85] 中删除 82 的问题。

由于 2) 转化为 1) 非常容易, 故只讨论 1)。从一个包含多于最少数目元素 (如果树叶同时是根节点, 那么最少元素数目是 1, 如果不是根节点, 则为  $\lceil m/2 \rceil - 1$ ) 的树叶中删除一个元素, 只需要将修改后的节点写回。(如果该节点是根节点, 则 B-树就成为空树)。从图 11-27a 的 B-树中删除 50, 需将修改后的节点 [20, 30, 40, 60, 70] 写回磁盘。而从图 11-27c 的 2-3 树中删除 85, 需将节点 [82] 写回磁盘。两种情况下, 在沿搜索路径到树叶的过程中都需要  $h$  次磁盘访问, 将包含被删除元素的修改后的树叶写回磁盘还需要一次额外的磁盘访问。

当被删除元素在一个非根节点中且该节点中的元素数量为最小值时, 可用其最相邻的左或右兄弟中的元素来替换它。注意到除了根节点以外的每个节点都会有一个最相邻的左兄弟或一个最相邻的右兄弟, 或二者都有。例如, 假设希望从图 11-27b 的 B-树中删除元素 25, 此次删除留下了一个节点 [20, 30], 它恰好有两个元素, 但是, 由于这个节点是七序 B-树的一个非根节点, 它必须至少要有三个元素, 而它的最相邻的左兄弟 [2, 3, 4, 6] 中多一个元素, 因此可把该节点中的最大元素移到其父节点中, 所牵涉的元素 (关键值为 10) 被向下移动, 从而产生图 11-28a 所示的 B-树。磁盘访问次数是 2 (从根到包含 25 的树叶) + 1 (读取该树叶的最相邻左兄弟) + 3 (写回修改后的树叶、兄弟和父节点) = 6。

假如没有检查 [20, 30] 的最相邻左兄弟, 而是检查它的最相邻右兄弟 [50, 60, 70]。由于此节点只含有三个元素, 所以不能从中删除元素。(若此节点有四个或更多的元素, 则可以把其中最小元素移到它的父节点中, 并且将这两个相邻兄弟之间的父节点中的元素移动到缺少一个元素的树叶中)。现在, 可以检查 [20, 30] 的最相邻左兄弟。执行检查需要一次额外的磁盘访问, 并且不能肯定在这个最相邻兄弟中有这样一个额外元素。为保持低次数的磁盘访问, 只检查缺少一个元素的最相邻兄弟之中的一个。

当最相邻兄弟中不含额外的元素时, 将两个兄弟与父节点中介于两个兄弟之间的元素合并成一个节点。由于两兄弟分别有  $d-2$  和  $d-1$  个元素, 合并后节点共有  $2d-2$  个元素。当  $m$  是奇数时,  $2d-2$  等于  $m-1$ ; 而当  $m$  是偶数时,  $2d-2$  等于  $m-2$ 。节点中有足够的空间来容纳这么多元素。

在本例中, 两兄弟 [20, 30] 和 [50, 60, 70] 以及关键值为 40 的元素被合并成一个节点 [20, 30, 40, 50, 60, 70]。得到的 B-树如图 11-27a 所示。删除过程中, 到达节点 [20, 25, 30] 需要 2 次磁

盘访问，读取最相邻右兄弟需要1次，将两个修改后的节点写回还需要2次，因此磁盘访问次数一共是5次。

由于合并减少了父节点中的元素个数，父节点有可能会缺少一个元素，如果这样，需要检查父节点的最相邻兄弟，要么从中取一个元素，要么与它合并。如果从最相邻右（左）兄弟中取一个元素，那么此兄弟节点的最左（最右）子树也将被读取到。如果进行合并，那么祖父节点也可能会缺少一个元素，此过程又需要在祖父节点中重复应用。最坏情况下，这种过程会一直回溯到根节点。当根节点缺少一个元素时，它变成空节点，将被抛弃，树的高度减1。

假设需要从图11-26的2-3树中删除10。删除留下了一个不含元素的树叶节点，它的最相邻右兄弟没有额外的元素。因此，两个兄弟树叶及父节点（10）中的元素被合并到一个节点中，新的树结构如图11-28b所示。现在第二层中有一个节点缺少一个元素，它的最相邻右兄弟中有一个额外的元素，最左边元素（关键值为50）移到父节点中，并将关键值为30的元素移下来，得到的2-3树如图11-28c所示。注意到[50,60]的左子树也被移动。到达包含被删除元素的树叶时执行了3次读访问，到达第二和三层的最相邻右兄弟节点时执行了2次读访问，将第一，二和三层的4个修改后的节点写回磁盘需要4次写访问。因此总的磁盘访问次数是9次。

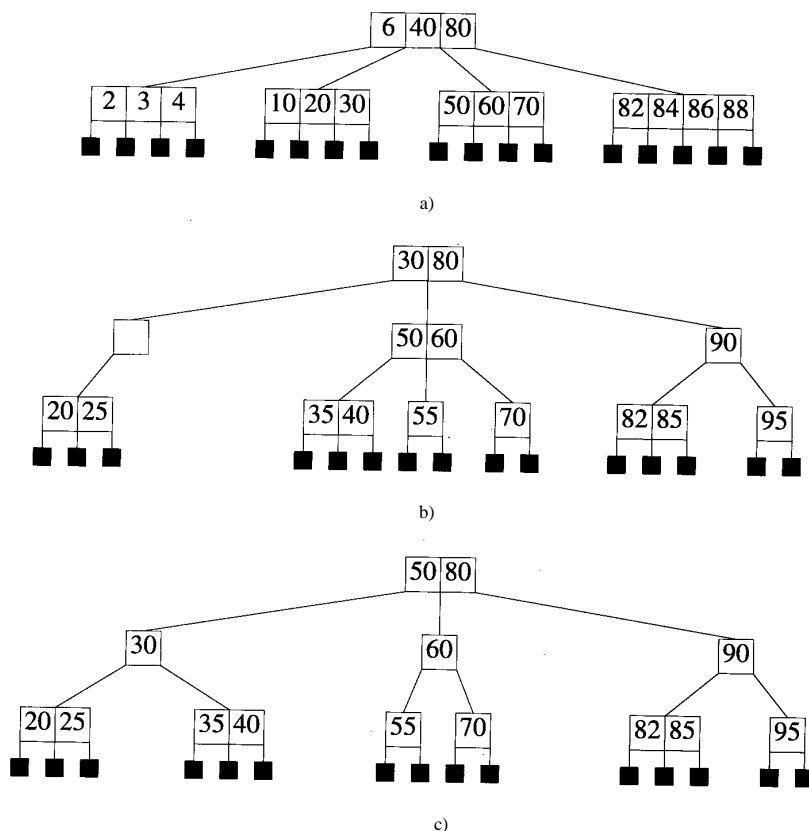
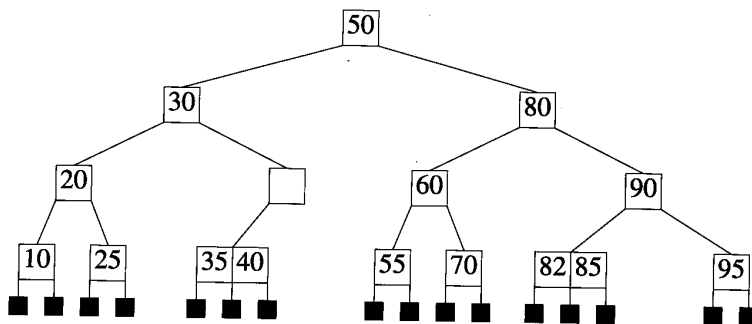


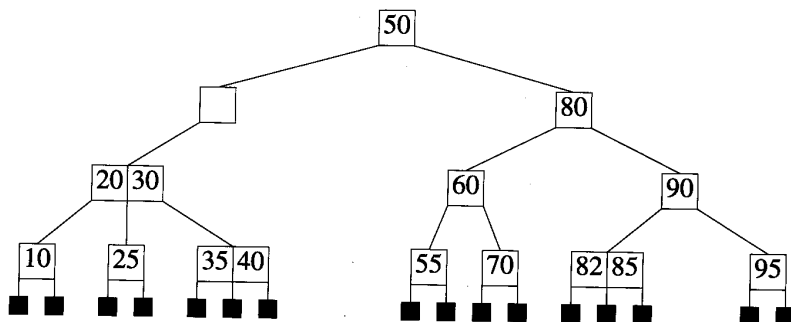
图11-28 B-树的删除

a) 从图11-27b中删除25 b) 树叶层合并之后 c) 从图11-26中删除10

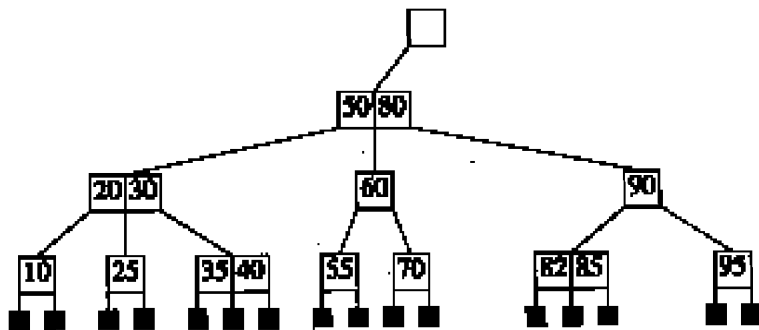
最后一个例子，考察删除图11-27c的2-3树中的44。当将44从树叶中删除时，树叶中将缺少一个元素，它的最相邻左兄弟没有额外的元素，因此两兄弟与父节点中的元素被合并，得到



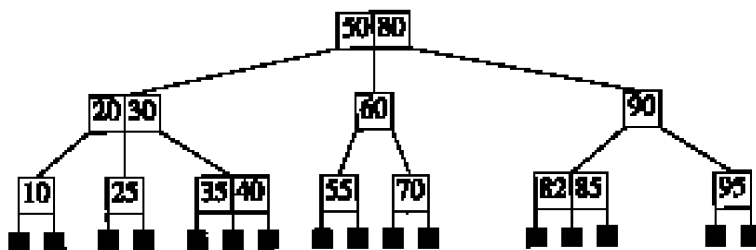
a)



b)



c)



d)

图11-29 从图11-27c 中的2-3树中删除44

a) 树叶层合并之后   b) 第3层合并之后   c) 第2层合并之后   d) 删除根节点之后

图11-29a 所示的树。现在第三层有一个节点，缺少一个元素，它的最相邻左兄弟中不含有额外的元素，因此两兄弟与父节点的元素合并，得到的树如图11-29b 所示，现在第二层中的一个节点缺少一个元素，它的最相邻右兄弟不含有额外的元素，执行合并后得到图11-29c 中的树。此时根节点缺少一个元素，由于只有根节点为空时它才会缺少一个元素，因此将根节点抛弃。最后的2-3树如图11-29d 所示，根节点被抛弃后，树的高度减少了一层。

找到含有被删除元素的树叶需要4次磁盘访问，对最相邻兄弟有3次访问，另有3次写访问，因此总的访问次数是10次。

对于高度为 $h$ 的B-树的删除操作，最坏情况出现在当合并发生在 $h, h-1, \dots, 3, 2$ 层时，需要从最相邻兄弟中获取一个元素。最坏情况下磁盘访问次数是 $3h = (\text{找到包含被删除元素需要 } h \text{ 次读访问}) + (\text{获取第2至 } h \text{ 层的最相邻兄弟需要 } h-1 \text{ 次读访问}) + (\text{在第3至 } h \text{ 层的合并需要 } h-2 \text{ 次写访问}) + (\text{对修改过的根节点和第2层的两个节点进行3次写访问})$ 。

#### 11.4.8 节点结构

在以上的讨论中假定节点的结构如下：

$$s, c_0, (e_1, c_1), (e_2, c_2), \dots, (e_s, c_s)$$

其中 $s$ 是节点中元素的个数， $e_i$ 是按关键值升序排列的元素， $c_i$ 是孩子指针。当元素的大小比关键值的大小更大时，可采用以下的节点结构：

$$s, c_0, (k_1, c_1, p_1), (k_2, c_2, p_2), \dots, (k_s, c_s, p_s)$$

其中 $k_i$ 是元素的关键值， $p_i$ 是相应元素在磁盘中的位置。利用这种结构，可以得到更高序的B-树，这种更高序的B-树称作B'-树。如果非树叶节点中不含有 $p_i$ 指针并且在树叶中用 $p_i$ 指针替换了空孩子指针，就可能产生更高序的B-树。

另一种可能是用平衡的二叉搜索树描述每个节点的内容。利用平衡的二叉搜索树可以减小B-树的序值，因为对于每个元素都需要一个左-右孩子指针以及一个平衡因子或颜色域。但是将元素插入到节点中或从节点中删除一个元素所花费的CPU时间减少了。这种方法能否导致性能的提高取决于具体的应用。在某些情况中，较小的 $m$ 可能增加B-树的高度，导致每一个搜索/插入/删除操作需要更多的磁盘访问。

#### 练习

31. 如果每个节点占用2个磁盘块并且需要2次磁盘访问才能搜索出来，那么在一棵 $2m$ 序B-树的搜索过程中需要的最大磁盘访问次数是多少？将该次数与节点大小占用1个磁盘块的 $m$ 序B-树的磁盘访问次数相比较，并论述节点大小大于磁盘块大小时的优点。

32. 删除一个 $m$ 序B-树中非树叶节点的元素需要的最大磁盘访问次数是多少？

33. 假设按如下方法修改从B-树中删除元素的方式：如果一个节点既有最相邻左兄弟也有最相邻右兄弟，那么在合并前对两个兄弟都要作检查。从一棵高度为 $h$ 的B-树中删除元素时需要的最大磁盘访问次数是多少？

\*34. 一棵2-3-4树可以描述为一棵二叉树，其中每一个节点要么是红色的，要么是黑色的。在2-3-4树中只含有一个元素的节点被描述为黑色节点；含有2个元素的节点被描述为有1个红色孩子的黑色节点（红色孩子既可以是黑色节点的左孩子，也可以是右孩子）；含有3个元素的节点被描述为有2个红色孩子的黑色节点。

1) 画一棵2-3-4树，其中至少包含一个2元素节点和一个3元素节点，用上述方法将其画成

带颜色节点的二叉树。

2) 验证该二叉树是一棵红-黑树。

3) 证明当用上述方法把任意二叉树描述成一棵带颜色的二叉树时，所得到的二叉树是一棵红-黑树。

4) 证明可以用相反的映射方式把任意一棵红-黑树描述成一棵2-3-4树。

5) 验证下列事实：对于红-黑树的插入操作，11.4.4节给出的改变颜色和旋转的方法，也可以从采用4)中映射模式的B-树插入方法中得到。

6) 对从红-黑树中删除元素的情况重做5)。

\*35. 设计一个类TwoThree，实现一棵2-3树。类中应包括搜索、插入和删除操作。测试代码是否正确。

\*36. 设计一个类TwoFour，实现一棵2-3-4树。类中应包括搜索、插入和删除操作。测试代码是否正确。

## 11.5 应用

### 11.5.1 直方图

在直方图问题中，从一个具有  $n$  个关键值的集合开始，要求输出不同关键值的列表以及每个关键值在集合中出现的次数（频率）。图11-30给出了一个含有10个关键值的例子。图11-30a给出了直方图的输入，直方图的表格形式如图11-30b所示，直方图的图形形式如图11-30c所示。直方图一般用来确定数据的分布，例如，考试的分数、图象中的灰色比例、在生产商注册的汽车和居住在洛杉矶的人所获得的最高学位都可以用直方图来表示。当关键值为从0到 $r$ 范围内的整数且 $r$ 的值足够小时，可以在线性时间内，用一个相当简单的过程（见程序11-6）产生直方图。在该过程中用数组元素 $h[i]$ 代表关键值 $i$ 的频率。可以使用程序11-6把其他关键值类型映射到这个范围中。例如，如果关键值是小写字母，则可以用映射 $[a,b,c,\dots,z]=[0,1,\dots,25]$ 。

$n = 10$ ; 关键值 = [2, 4, 2, 2, 3, 4, 2, 6, 4, 2]

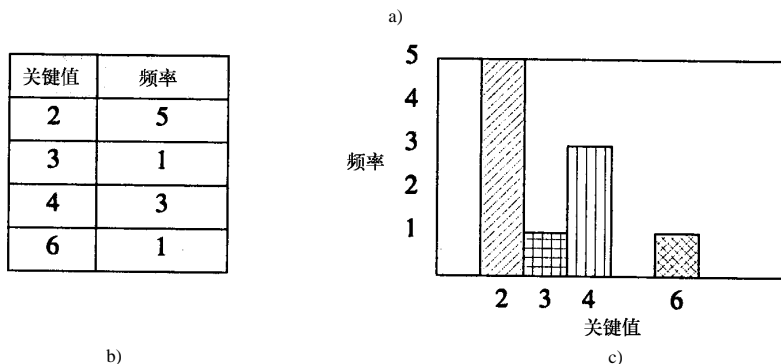


图11-30 直方图举例

a) 输入 b) 表格形直方图 c) 直方图的图形形式

程序11-6 简单的直方图程序

```
void main(void)
```

```

{// 非负整数值的直方图
int n, // 元素数
    r; // 整数值位于 0 到 r 之间
cout << "Enter number of elements and range" << endl;
cin >> n >> r;

// 创建数组 h
int *h;
try {h = new int[r+1];}
catch (NoMem)
    {cout << "range is too large" << endl;
     exit(1);}

// 将数组h初始化为0
for (int i = 0; i <= r; i++)
    h[i] = 0;

// 输入数据并计算直方图
for (i = 1; i <= n; i++) {
    int key; // input value
    cout << "Enter element " << i << endl;
    cin >> key;
    h[key]++;
}

// 输出直方图
cout << "Distinct elements and frequencies are" << endl;
for (i = 0; i <= r; i++)
    if (h[i]) cout << i << " " << h[i] << endl;
}

```

当关键值类型不是整型（如关键值类型是实数）或关键值范围变化很大时，程序 11-6 不可用。假设要确定一个文本中不同词语出现的频率，与文本中实际出现的数量相比，可能的不同词语的数量是非常大的，在这种情况下，可以将关键值排序，然后用一个简单的自左至右的扫描方法确定每一个不同关键值的数量。搜索可在  $O(n \log n)$  时间内完成（例如，用 HeapSort（见程序 9-12）堆排序），从左至右扫描需要  $\Theta(n)$ ；因此总的复杂性是  $O(n \log n)$ 。当与  $n$  相比，不同关键值的数量  $m$  非常小时，可以进一步改进这种方法。通过使用 AVL 和红-黑树之类的二叉搜索树，可以在  $O(n \log m)$  时间内解决直方图问题。另外，采用平衡的搜索树只需把不同的关键值存储在内存中，因此，当  $n$  的值非常大，没有足够的内存来容纳所有的关键值（当然，对于不同关键值来说，内存是足够的）时，这种方法是适用的。

上述方法使用的是二叉搜索树，其平均复杂性为  $O(n \log m)$ 。通过用平衡的搜索树来取代二叉搜索树，可以得到所要求的复杂性。在二叉搜索树方案中，我们扩充了类 BSTree，增加了以下共享成员：

```

BSTree <E,K>& InsertVisit
    (const E&e, void(*Visit) (E& u));

```

若树中不存在关键值等于  $e.key$  的元素时，该函数将元素  $e$  插入到搜索树中。如果存在这样的元素  $u$ ，则调用函数 `Visit`。为了获得 `InsertVisit` 的代码，可把 `Insert`（见程序 11-3）中的如下语句：

```
else throw BadInput( );
```

用下面的语句来替换：

```
else {Visit(p data);
      return *this;};
```

程序 11-7 给出了新的直方图程序代码。在访问一个元素的过程中，该元素的频率增加了 1。

程序 11-7 使用搜索树的直方图程序

---

```
class eType {
    friend void main(void);
    friend void Add1(eType&);
    friend ostream& operator <<(ostream&, eType);
public:
    operator int() const {return key;}
private:
    int key, // 元素值
        count; // 频率
};

ostream& operator<<(ostream& out, eType x)
{out << x.key << " " << x.count << " "; return out;}

void Add1(eType& e) {e.count++;}

void main(void)
{// 使用搜索数的直方图程序
    BSTree<eType,int> T;
    int n; // 元素数目
    cout << "Enter number of elements" << endl;
    cin >> n;

    // 输入元素并插入树中
    for (int i = 1; i <= n; i++) {
        eType e; // 输入元素
        cout << "Enter element " << i << endl;
        cin >> e.key;
        e.count = 1;
        // 将e 插入树中，除非已存在同值元素
        // 在后一种情况下，将count 增1
        try {T.InsertVisit(e, Add1);}
        catch (NoMem)
            {cout << "Out of memory" << endl;
             exit(1);}
    }

    // 输出所有相异的元素以及它们的计数
```



```
cout << "Distinct elements and frequencies are"
    << endl;
T.Ascend();
}
```

### 11.5.2 用最优匹配法求解箱子装载问题

将 $n$ 个物品装入到容量为 $c$ 的箱子中的最优匹配方法，已在10.5.1节中介绍过。通过使用平衡的搜索树，能够在 $O(n \log n)$ 时间内完成箱子装载过程。搜索树的每一个元素代表一个正在使用的并且还能继续存放物品的箱子。假设当物品 $i$ 被装载时，已使用的九个箱子中还有一些剩余空间，设这些箱子的剩余容量分别是1,3,12,6,8,1,20,6和5。可以用一棵二叉搜索树来存储这九个箱子，每个箱子的剩余容量作为节点的关键值，因此这棵树应是允许有重复值的二叉搜索树（即为DBSTree或DAVLtree）。

图11-31给出了一棵存储上述九个箱子的二叉搜索树。节点内部是箱子的剩余容量，节点外侧是箱子的名称。这棵树也是一棵AVL树。如果需要装载的物品 $i$ 需要 $s[i]=4$ 个空间单元，那么可以从根节点开始搜索，直至找到最优匹配的箱子。由根节点可知，箱子 $h$ 的剩余容量是6，由于物体 $i$ 可以放入该箱中，因此箱子 $h$ 成为一个候选。由于根节点右子树中所有箱子的剩余容量至少是6，故不需要再从右子树中寻找合适的箱子，寻找只需要在左子树中进行。箱子 $b$ 的剩余容量不能容纳该物品，因此搜索转移到了箱子 $b$ 的右子树中，右子树的根节点箱子 $i$ 可以容纳该物品，所以箱子 $i$ 成为适合的候选。从这里，搜寻转移到箱子 $i$ 的左子树，由于左子树为空，因此不再有更好的候选，所以箱子 $i$ 即要找的箱子。

再看另一个例子，假设 $s[i]=7$ ，从根节点开始搜寻。根节点的箱子 $h$ 不能装载物品 $i$ ，因此转移到右子树中，箱子 $c$ 可以容纳物品 $i$ ，因此成为新的候选箱子。从这里再向下搜寻，节点 $d$ 没有足够的剩余容量容纳此物品，因此继续查找 $d$ 的右子树，箱子 $e$ 可以容纳物品 $i$ ，因此 $e$ 成为新的候选，然后转移到 $e$ 的左子树，左子树为空，搜索终止。

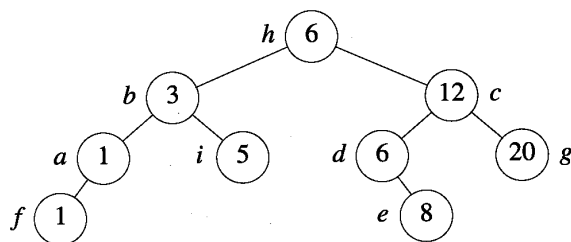


图11-31 含有重复值的AVL树

当找到最合适的箱子后，可以将它从搜索树中删除，将其剩余容量减去 $s[i]$ ，再将它重新插入到树中（除非它的剩余容量为零）。如果没有找到合适的箱子，则可以用一个新的箱子来装载物品 $i$ 。

为了实现上述思想，既可以采用类DBSTree（可以得到平均性能 $O(n \log n)$ ），也可以采用类DAVLtree（在各种情况中都能得到平均性能 $O(n \log n)$ ）。无论哪一种方法，都需要扩充类的定义，增加共享成员FindGE(k, Kout)，该函数可以找到剩余容量Kout  $\geq k$ 的具有最小剩余容量的箱子。FindGE的代码见程序11-8，它的复杂性是 $O(\text{height})$ 。类AVLtree中FindGE的代码可以与程序11-8完全相同。

程序11-8 寻找大于等于K的最小关键值

```
template<class E, class K>
```

```

bool DBSTree<E,K>::FindGE(const K& k, K& Kout) const
{
    // 寻找值 k 的最小元素
    BinaryTreeNode<E> *p = root, // 搜索指针
                      *s = 0;    // 指向迄今所找到的 >= k 的最小元素

    // 对树进行搜索
    while (p) {
        // p 是一个候选吗?
        if (k <= p->data) { // 是的
            s = p; // p 是比 s 更好的候选
            // 较小的元素仅会在左子树中
            p = p->LeftChild;
        }
        else // 不是, p->data 太小, 试一试右子树
            p = p->RightChild;
    }

    if (!s) return false; // 没找到
    Kout = s->data;
    return true;
}

```

程序 11-9 使用最优匹配方法将  $n$  个物品装入到箱子中，它使用了与 FirstFit（见程序 10-7）相同的接口。这样，只要用 `# include` 语句将类 DBSTree 的文件加载到程序中，就可以用程序 10-6 来调用程序 11-9。

程序 11-9 用最优匹配法求解箱子装载问题的算法

```

class BinNode {
    friend void BestFitPack(int *, int, int);
    friend ostream& operator<<(ostream&, BinNode);
public:
    operator int() const {return avail;}
private:
    int ID, // 箱子标号
        avail; // 可用容量
};

ostream& operator<<(ostream& out, BinNode x)
{
    out << "Bin " << x.ID << " " << x.avail;
    return out;
}

void BestFitPack(int s[], int n, int c)
{
    int b = 0; // 所使用的箱子数
    DBSTree<BinNode, int> T; // 由箱子容量构成的树

    // 依次装载每个物品
    for (int i = 1; i <= n; i++) { // 装载物品 i
        int k; // 最优匹配的箱子
        BinNode e; // 对应的节点
    }
}

```

```

if (T.FindGE(s[i], k)) // 寻找最优箱子
    T.Delete(k, e); // 从树中删除最优箱子
else { // 没有足够大的箱子
    // 从一个新箱子开始
    e = *(new BinNode);
    e.ID = ++b;
    e.avail = c;

    cout << "Pack object " << i << " in bin " << e.ID << endl;

    // 更新可用容量并将箱子插入树中，除非可用容量为 0
    e.avail -= s[i];
    if (e.avail) T.Insert(e);
}
}

```

### 11.5.3 交叉分布

在交叉分布问题中，从一个布线通道开始，通道的顶部和底部各有  $n$  个针脚。图 11-32 给出了  $n=10$  的情况。布线区域是图中带阴影的长方形区域，针脚的序号从 1 到  $n$ ，在通道的顶部和底部从左至右分布。另外，有  $[1, 2, 3, \dots, n]$  的一个排列  $C$ 。必须用一根电线将顶部的针脚  $i$  与底部的针脚  $C_i$  连接起来。在图 11-32 的例子中， $C = [8, 7, 4, 2, 5, 1, 9, 3, 10, 6]$ 。需要连接的  $n$  根线被编号为 1 到  $n$ ，第  $i$  根线连接顶部的  $i$  和底部的  $C_i$ 。当且仅当  $i < j$  时，连线  $i$  在连线  $j$  的左边。

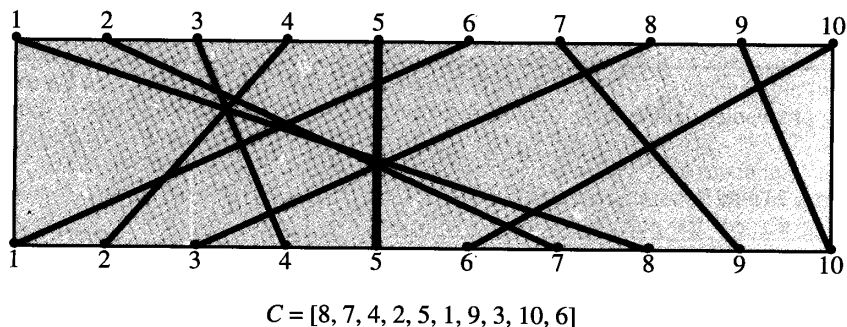


图 11-32 布线举例

图 11-32 显示了在布线区域中无论线路 9 和 10 如何设置，它们一定会在某一点相交。我们不希望交叉的出现，以免出现短路，为此可以在交叉点放置绝缘物或将两条线路布设在不同层。因此，必须寻找最小的交叉点数目。可以验证，当线路是按图 11-32 中的直线布设时，交叉点的数量是最少的。

每个交叉点用  $(i, j)$  表示， $i$  和  $j$  是通过该交叉点的两条线路。识别所有交叉点的一种方法是检查线路的每一个  $(i, j)$  并且检验其中两条直线是否相交。为避免对同一交叉点检验两次，可以要求  $i < j$ （即交叉点  $(10, 9)$  与  $(9, 10)$  是一样的）。设  $k_i$  是  $(i, j)$  的数量， $i < j$ 。在图 11-32 的例子中， $k_9 = 1$ ， $k_{10} = 0$ 。在图 11-33 中，列出了图 11-32 中所有的交叉及  $k_i$  的值。该表的第  $i$  行首先给出了  $k$  的值，然后给出了  $j$  的值， $i < j$ ，其中  $i$  与  $j$  是相交的，相交的总数  $K$  由所有  $k_i$  的和确定。在本例中  $K = 22$ 。由于  $k_i$  只记录了线路  $i$  与其右边线路（即  $i < j$ ）相交的数量，因此  $k_i$  给

出的是线路 $i$ 右侧相交的数量。

为了降低复杂性,要求在通道的每一半区域中都含有大约相同的交叉点(一部分含 $k/2$ 个交叉点,另一部分含 $\lceil k/2 \rceil$ 个交叉点)。图11-34给出了图11-32中的另一种布线方法,在这种方法中,在通道的上、下部分各有11个交叉点。

上半部分的连接由排列 $A=[1,4,6,3,7,2,9,5,10,8]$ 给出,即顶部的针脚 $i$ 与中间的针脚 $A_i$ 相连。下半部分的连接由排列 $B=[8,1,2,7,3,4,5,6,9,10]$ 给出,中间的针脚 $i$ 与底部的针脚 $B_i$ 相连。可以看出 $C_i=B_{A_i}$ ,  $1 \leq i \leq n$ 。要完成 $C$ 给出的连接,这个等式是必不可少的。

通过检查 $(i, j)$ ,在 $\Theta(n^2)$ 的时间内可以计算出相交数量 $k_i$ 及总的相交数量 $K$ 。可采用程序11-10中的线性表从 $C$ 计算出 $A$ 和 $B$ 。

$i$	$k_i$	交叉									
1	7	2	3	4	5	6	8	10			
2	6	3	4	5	6	8	10				
3	3	4	6	8							
4	1	6									
5	2	6	7								
6	0										
7	2	8	10								
8	0										
9	1	10									
10	0										

图11-33 相交表

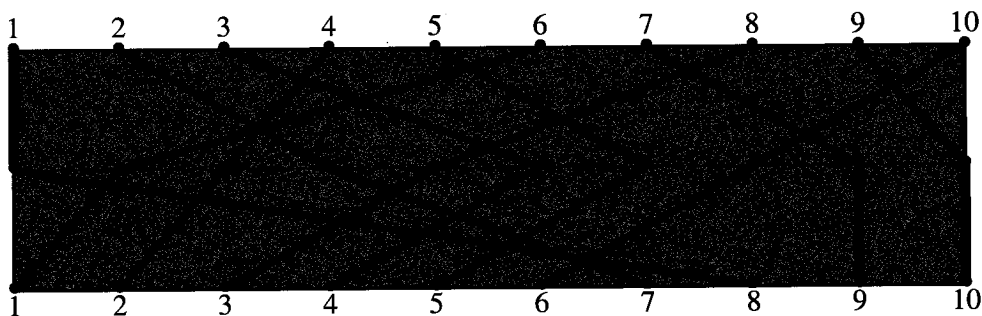


图11-34 分割交叉

程序11-10 使用线性表的交叉分布程序

```

LinearList<int> L(n);
int r = K/2; // 上半部分需要的交叉数

// 从右至左扫描线路
int w = n; // w 是当前线路
while (r) { // 上半部分需要更多的交叉
    if (k[w] < r) { // 使用w的所有交叉 L.Insert(k[w], w);
        r -= k[w];
    } else { // 仅使用w的r个交叉
        L.Insert(r, w);
        r = 0;
    }
    w--;
}

// 在中线上确定线路排列
// 前w条线的排列次序不变
for (int i = 1; i <= w; i++)
    X[i] = i;

```

```
// 其余线路的排列次序来自 L
```

```
for (i = w+1; i <= n; i++)
```

```
    L.Delete(1, X[i]);
```

```
// 计算上半部分的排列
```

```
for (i = 1; i <= n; i++)
```

```
    A[X[i]] = i;
```

```
// 计算下半部分的排列
```

```
for (i = 1; i <= n; i++)
```

```
    B[i] = C[X[i]];
```

在while 循环中，从右至左扫描这些线路，确定它们在布线通道中央的相关次序。目标是在通道中部产生一个布线次序，使得在布线通道上半部分的中间正好有  $r=k/2$  个交叉。

线性表L用于记录通道中部当前所得到的次序。当考察线路  $w$  时，把  $w$  右侧与  $w$  相交的线路中的最多  $k[w]$  个交叉分配到上半部分。第一个交点是与  $L$  中的第一条线路相交产生的，第二个交点是与  $L$  中第二条线路相交产生的，如此往复。如果将与  $w$  相交的  $k[w]$  条线路中的  $c$  条分配到上半部分，那么这条线路必定与  $L$  中的前  $c$  条线路相交。另外从  $w$  到  $n$  的次序也可以通过将  $w$  插入到  $L$  中的第  $c$  条线之后而得到。当在程序 11-10 的while 循环中考察线路  $w$  时，线路  $w+1$  至  $n$  已经在  $L$  中。而且，由于  $k[w]$  不会超过线路  $w$  右边的线路数，因此在考察  $w$  时， $L$  中至少已有  $k[w]$  条线路。

当在程序 11-10 中的while 循环中考察线路  $w$  时，除非上半部分所需要的相交数  $r$  比  $k[w]$  小，否则将所有右边相交点都分配到上半部分。

当while 循环终止时，布线通道中间的线路次序就建立好了。线路 1 到  $w$  在上半部分没有相交，因此在这半部分中不必改变它们的相对次序。因而， $X[1:w]=[1,2,\dots,w]$ ，余下的线路排序由线性表  $L$  给出，程序 11-10 中的前两个for 循环构造了  $X$ 。

下面根据图 11-32 的例子来构造  $X$ 。先将线路 10 添加到  $L$  中，得  $L=(10)$ 。没有产生交叉，所以接下来把线路 9 添加到  $L$  中，得  $L=(10,9)$ 。这时在上半部分产生了 1 个相交。之后将 8 加入到第  $k_8$  个元素后边，得  $L=(8,10,9)$ 。此时上半部分的右边相交总数仍然是 1。下面将 7 加入到  $L$  的第二个元素之后，上半部分有 2 个相交， $L$  变为  $(8,10,7,9)$ ，所需的相交次数  $r$  降到了 8。当 6 加入后，得到  $L=(6,8,10,7,9)$ ， $r=8$ 。将线路 5 加入后又产生了 2 个相交， $L=(6,8,5,10,7,9)$ ， $r=6$ 。当 4 被加到  $L$  的第一个元素之后时，产生了一个相交，得到  $L=(6,4,8,5,10,7,9)$ ， $r=5$ 。下面将 3 加入，得到  $L=(6,4,8,3,5,10,7,9)$ ， $r=2$ 。最后，考察 2 时，我们发现，虽然它能产生  $k_2=6$  个相交，但只能将其中的 2 个分配到通道的上半部分，因此它被插入到  $L$  第二个元素的后边，得到  $L=(6,4,2,8,3,5,10,7,9)$ 。剩下的线路保持它们的相对次序。

现在完成了上半部分的布线，然后计算线路的排列，通过在序列  $(1,2,\dots,w)$  上附加  $L$  得到  $X=[1,6,4,2,8,3,5,10,7,9]$ 。

排列  $A$  与  $X$  关系密切。 $A[j]$  表明线路  $j$  应该连接到中间的哪个针脚上，而  $X[j]$  表明哪一条线路连接到中间的针脚  $j$  上。程序 11-10 中的第三个for 循环就是用这种信息来计算  $A$  的。在第四个for 循环中，利用  $X$  和  $C$  计算出  $B$ 。

由于将元素插入到大小为  $s$  的线性表中需要的时间是  $O(s)$ ，所以程序 11-10 中的while 循环需占用  $O(n^2)$  时间，第二个for 循环也需要这么多时间，其他代码需要  $\Theta(n)$  时间，因此程序 11-10 的全部代码复杂度是  $O(n^2)$ 。将程序 11-10 需要的时间与计算  $K$  和  $k[i]$  所需要的时间联系在一起，

可知在使用线性表解决交叉分布问题时，所需要的全部时间是  $O(n^2)$ 。

通过使用平衡的搜索树来代替线性表，可以把解决方案的复杂性降低到  $O(n \log n)$ 。为了得到平均复杂性  $O(n \log n)$ ，可以使用带索引的二叉搜索树，而不使用带索引的平衡搜索树。这两种情况在技术上是相同的，下面将用带索引的二叉搜索树来阐述此技术。

首先，来看一下如何计算相交数量  $k_i, 1 \leq i \leq n$ 。假设检查的线路次序是  $n, n-1, \dots, 1$  并且检查线路  $i$  时，将  $C_i$  插入到带索引的搜索树中。对于图 11-32 中的例子，首先从空树开始。检查线路 10 并将  $C_{10} = 6$  插入到空树中，得到图 11-35a 所示的树。节点外侧的数字是其 LeftSize 的值，节点内部是其关键值（或 C 的值）。注意到  $k_n$  总是为零，因此设  $k_n = 0$ 。然后检查线路 9 并将  $C_9 = 10$  插入到树中，得到图 11-35b 所示的树。为了实现这次插入，越过了根节点（其 LeftSize=1）。根据这个 LeftSize 值，可知线路 9 的底部节点正好在目前所看到的一条线路的右边，因此  $k_9 = 1$ 。下面检查线路 8，将  $C_8 = 3$  插入到树中，得到图 11-35c 所示的树。由于  $C_8$  是树中最小的入口，因此没有线路相交且  $k_8 = 0$ 。对于线路 7， $C_7 = 9$  被插入后得到图 11-35d 中的树， $C_7$  成为树中的第三个最小入口。通过对进入其右子树的各节点的 LeftSize 值连续求和，可以确定  $C_7$  是第三个最小入口。 $C_7$  被插入时，这个和是 2，因此，新元素是当前第三个最小的。由此可以得出结论，它的底部节点位于树中其他 2 个节点的右边，因此， $k_7 = 2$ 。按此方法进行下去，当检查线路 6 至 2 时，产

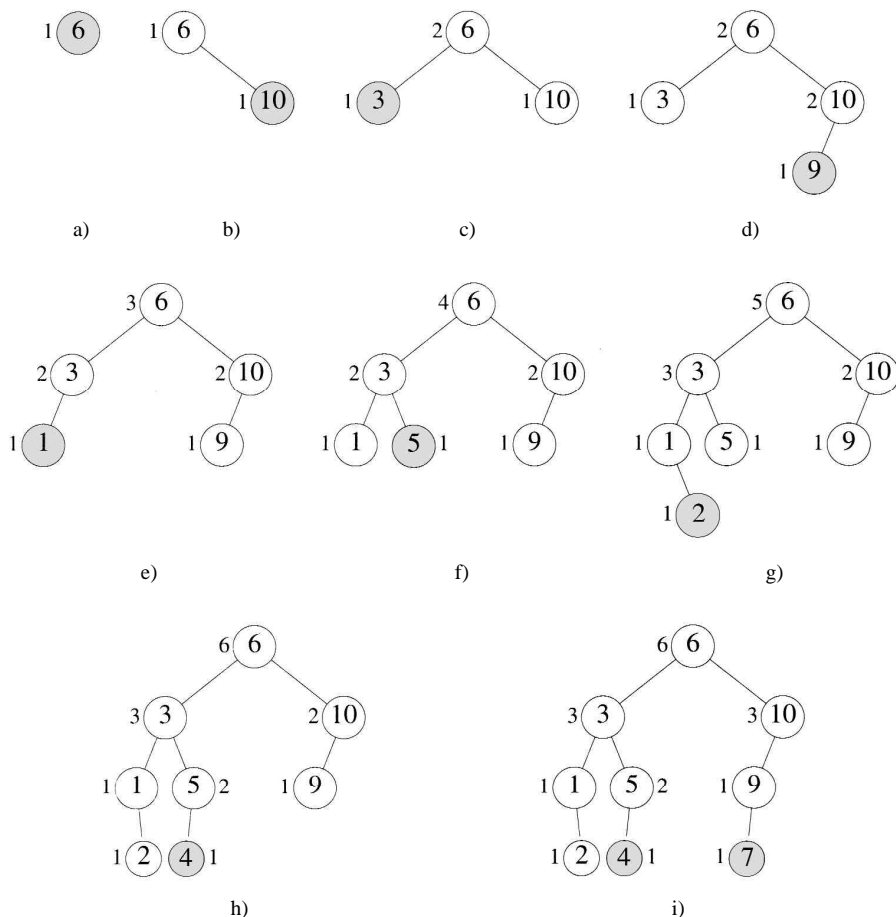


图 11-35 计算交叉的数量



生了图11-35e至图11-35i的树。最后，检查线路1时，将 $C_1=8$ 插入到树中，作为关键值为7的节点的右孩子。进入其右子树的各节点的LeftSize值之和是 $6+1=7$ ，线路1的底部节点位于树中7条线路的右边，因此 $k_1=7$ 。

检查线路 $i$ 和计算 $k_i$ 需要的时间是 $O(h)$ ，其中 $h$ 是当前带索引的搜索树的高度。因此，用带索引的二叉搜索树可在 $O(n \log n)$ 平均时间内或用带索引的平衡搜索树在 $O(n \log n)$ 平均时间内计算出所有的 $k_i$ 。

为计算 $A$ ，可以用一个修改的带索引的搜索树来实现程序11-10的代码。在这种修改中，元素没有关键值，每个元素只有一个位置或阶。Insert( $j, e$ )用来插入元素 $e$ 使得 $e$ 成为阶为 $j+1$ 的元素。为了按阶的次序排列元素，可以进行一次中序遍历。执行程序11-10所需要的时间为 $O(n \log n)$ 。

得到排列 $A$ 的另一种方法是首先计算 $r = \sum_{i=1}^n k_i$ 和 $s$ =最小的 $i$ ，使得 $\sum_{l=i}^n k_l \geq r$ 。对于上面的例子有 $r=11$ 和 $s=3$ 。程序11-10实现了线路 $n, n-1, \dots, s$ 的所有相交，其中上半部分包含线路 $s-1$ 的 $r - \sum_{l=s}^n k_l$ 个相交，下半部分包含余下的相交。为了得到上半部分的相交点，对插入 $C_s$ 后的树进行检查。在本例中，检查图11-35h中的树。对树 $h$ 的中序遍历可产生序列(1,2,3,4,5,9,10)。用相应的线路编号来替换这些底部节点，可得到序列(6,4,8,3,5,10,7,9)，它给出了按图11-35h中所描述的9个相交线路所得到的排列。对于另外2个相交，将线路 $s=2$ 插入到该序列中第二条线路之后，得到新的线路序列(6,4,2,8,3,5,10,7,9)。剩下的线路1到 $s-1$ 加到序列前部，可得(1,6,4,2,8,3,5,10,7,9)，它就是程序11-10所计算出的排列 $X$ 。为了用这种方法得到 $X$ ，需要重新运行用来计算 $k_i$ 的部分代码，执行一个中序遍历，插入线路 $s$ 并在序列之前增加少量线路。所有步骤需要的时间是 $O(n \log n)$ 。程序11-10的最后两个for循环可在线性时间内从 $X$ 中得到 $A$ 和 $B$ 。

## 练习

37. 写一个直方图程序，首先将 $n$ 个关键值输入到一个数组中，然后对数组进行排序，最后对数组从左到右进行扫描，输出不同的关键值和每个关键值出现的次数。

38. 写一个直方图程序，使用链表散列而不是程序11-7中的二叉搜索树来储存不相同的关键值和它们的频率。将程序的运行时间与程序11-7相比较。

39. 1) 扩充类DBSTree，增加共享成员DeleteGE( $k, e$ )。该函数用来删除最小关键值不小于 $k$ 的元素，被删除元素在 $e$ 中返回，如果删除失败DeleteGE可以引发异常。

2) 用DeleteGE(而不是FindGE)设计一个BestFit的新版本。

3) 哪一种运行得更快？为什么？

40. 1) 用一个带索引的AVL搜索树为交叉分布问题设计一个性能为 $O(n \log n)$ 的算法。

2) 检验代码是否正确。

3) 根据实际运行时间，把该算法与本节(见程序11-10)所介绍的 $\Theta(n^2)$ 算法进行比较。可使用随机产生的排列 $C$ 和 $n=1000, 10\ 000$ 和 $50\ 000$ 来进行实验。

## 11.6 参考及推荐读物

AVL树是由G.Adelson-Velskii和E.Landis在1962年发明的，在D.Knuth. *The Art of Computer Programming: Sorting and Searching*. Addison-Wesley, 1973. 一书中可以找到更多的关于这些树的材料。



红-黑树是由R.Bayer在1972年发明的，但是Bayer将这种树叫作“对称的平衡B-树”，红-黑树这一术语是Guibas和Sedgewick在更详细地研究了这种树之后在1978年提出的。这方面的先期论文有R.Bayer. Symmetric Binary B-Tress: Data Structures and Maintenance Algorithms. *Acta Informatica*, 1, 1972, 290~306 和L.Guibas, R.Sedgewick. A Dichromatic Framework for Balanced Trees, *Proceedings of the 10th IEEE Symposium on Foundations of Computer Science*, 1978, 8~21。

用红-黑树来实现优先搜索树方面的应用可参考 E.McCreight. Priority Search Trees. *SIAM Journal on Computing*, 14, 2, 1985, 257~276。

交叉分布问题的解决算法由S.Cho和S.Sahni给出（未发表）。

具有相同渐进复杂性的各种不同搜索树结构可参考 E.Horowitz, S.Sahni, D.Mehta. *Fundamentals of Data Structures in C++*. W.H.Freeman, 1994。这本书还介绍了B'-树和其他各种变化的B\*-树。

## 第12章 图

恭喜！你已经成功穿越了“树”的森林，下面要学习图这种数据结构。令人惊叹的是，图可以用来描述成千上万的实际问题，不过，我们仅研究其中的一小部分。本章的主要内容如下：

- 图的若干术语：顶点，边，邻接，关联，度，回路，路径，连通构件，生成树。
- 图的三种类型：无向图，有向图和加权的图。
- 图的常用表示方法：邻接矩阵，邻接链表和邻接压缩表。
- 图的标准搜索方法：宽度优先搜索和深度优先搜索。
- 在图中寻找路径，在无向图中寻找连通构件以及在无向连通图中寻找生成树的算法。
- 如何把抽象数据类型表示成一个抽象类。

本章所使用的新的C++特征是：抽象类，虚函数和虚基类。

### 12.1 基本概念

简单地说，图（graph）是一个用线或边连接在一起的顶点或节点的集合。正式一点的说法是，图  $G=(V,E)$  是一个  $V$  和  $E$  的有限集合，元素  $V$  称为顶点（vertex，也叫作节点或点），元素  $E$  称为边（edge，也叫作弧或连线）， $E$  中的每一条边连接  $V$  中两个不同的顶点。可以用  $(i,j)$  来表示一条边，其中  $i$  和  $j$  是  $E$  所连接的两个顶点。

一般来说，图是由回路和边组成，如图12-1所示。在图12-1中有些边是带方向的（带箭头），而有些边是不带方向的。带方向的边叫有向边（directed edge），而不带方向的边叫无向边（undirected edge）。对无向边来说， $(i,j)$  和  $(j,i)$  是一样的；而对有向边来说，它们是不同的。前者的方向是从  $i$  到  $j$ ，后者是从  $j$  到  $i$ 。

当且仅当  $(i,j)$  是图中的边时，顶点  $i$  和  $j$  是邻接的（adjacent）。边  $(i,j)$  关联（incident）于顶点  $i$  和  $j$ 。图12-1a 中的顶点1和2是邻接的，顶点1和3，1和4，2和3，3和4也是邻接的，除此之外，这个图中没有其他邻接的顶点。边  $(1,2)$  关联于顶点1和2， $(2,3)$  关联于顶点2和3。

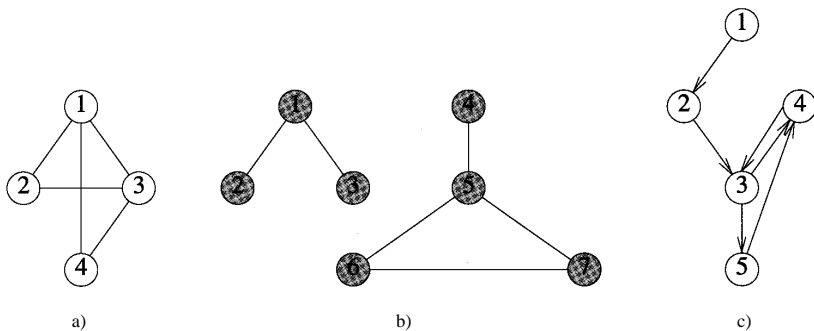


图12-1 图

◎ 有些书中用  $\{i,j\}$  表示无向边，而用  $(i,j)$  表示有向边。还有一些书用  $(i,j)$  表示无向边，用  $\langle i,j \rangle$  表示有向边。本书对两种边使用同一符号  $(i,j)$ ，边有向与否可从上下文中看出。

在有向图中, 有时候对邻接和关联的概念作更精确的定义非常有用。有向边  $(i, j)$  是关联至 (incident to) 顶点  $j$  而关联于 (incident from) 顶点  $i$ 。顶点  $i$  邻接至 (adjacent to) 顶点  $j$ , 顶点  $j$  邻接于 (adjacent from) 顶点  $i$ 。在图 12-1c 的图中, 顶点 2 邻接于顶点 1, 而 1 邻接至顶点 2。边  $(1, 2)$  关联于顶点 1 而关联至顶点 2。顶点 4 邻接至顶点 3 且邻接于顶点 3。边  $(3, 4)$  是关联于顶点 3 而关联至顶点 4。对于无向图来说, “至” 和 “于” 的含义是相同的。

如果使用集合的表示方法, 图 12-1 中的几个图可以用如下方法表示:  $G_1 = (V_1, E_1)$ ;  $G_2 = (V_2, E_2)$  和  $G_3 = (V_3, E_3)$ , 其中:

$$\begin{aligned} V_1 &= \{1, 2, 3, 4\}; & E_1 &= \{(1, 2), (1, 3), (2, 3), (1, 4), (3, 4)\} \\ V_2 &= \{1, 2, 3, 4, 5, 6, 7\}; & E_2 &= \{(1, 2), (1, 3), (4, 5), (5, 6), (5, 7), (6, 7)\} \\ V_3 &= \{1, 2, 3, 4, 5\}; & E_3 &= \{(1, 2), (2, 3), (3, 4), (4, 3), (3, 5), (5, 4)\} \end{aligned}$$

如果图中所有的边都是无向边, 那么该图叫作无向图 (undirected graph), 图 12-1a 和 b 都是无向图。如果所有的边都是有向的, 那么该图叫作有向图 (directed graph), 图 12-1c 是一个有向图。

由定义知道, 一个图中不可能包括同一条边的多个副本, 因此, 在无向图中的任意两个顶点之间, 最多只能有一条边。在有向图中的任意两个顶点之间, 最多只能有一条边从顶点  $i$  到顶点  $j$  或从  $j$  到  $i$ 。并且一个图中不可能包含自连边 (self-edge), 即  $(i, i)$  类型的边, 自连边也叫作环 (loop)。

通常把无向图简称为图, 有向图仍称为有向图 (digraph)。在一些图和有向图的应用中, 我们会为每条边赋予一个权或耗费, 这种情况下, 用术语加权有向图 (weighted graph) 和加权无向图 (weighted digraph) 来描述所得到的数据对象。术语网络 (network) 在这里是指一个加权有向图或加权无向图。实际上, 这里定义的所有图的变化都可以看作网络的一种特殊情况——一个无向 (有向) 图可以被看作是一个所有边具有相同权的无向 (有向) 网络。

## 12.2 应用

无向图, 有向图和网络常常用于电子网络的分析、化合物 (特别是碳氢化合物) 的分子结构研究、空中航线和通信网络的描述、项目策划、遗传研究、统计、社会科学及其它各种领域。这一节将用图来阐述一些实际问题。

例 12-1 [路径问题] 城市中有许多街道, 每一个十字路口都可以看作图中一个顶点, 邻接两个十字路口之间的每一段街道既可以看作一条, 也可以看作两条有向边。如果街道是双向的, 就用两条有向边。如果街道是单向的, 就用一条有向边。图 12-2 给出了假想的街道和相应的有向图。图中有三条街道: 街道 1, 街道 2 和街道 3 以及两条大街: 大街 1 和大街 2。十字路口用数字 1 到 6 进行编号, 相应的有向图 (如图 12-2b 所示) 的顶点标号与图 12-2a 给出的十字路口的标号相同。

当且仅当对于每一个  $j$  ( $1 \leq j \leq k$ ), 边  $(i_j, i_{j+1})$  都在  $E$  中时, 顶点序列  $P = i_1, i_2, \dots, i_k$  是图或有向图  $G = (V, E)$  中一条从  $i_1$  到  $i_k$  的路径。当且仅当相应的有向图中顶点  $i$  到顶点  $j$  有一条路径时, 十字路口  $i$  到  $j$  之间存在一条路径。在图 12-2b 的有向图中, 5, 2, 1 是从 5 到 1 的一条路径, 在这个有向图中, 从 5 到 4 之间没有路径。

简单路径是这样一条路径: 除第一个和最后一个顶点以外, 路径中其他所有顶点均不同。路径 5, 2, 1 是简单路径, 而 2, 5, 2, 1 则不是。

对于图或有向图的每一条边, 均可以给出一个长度。路径的长度是路径上所有边的长度之和。从十字路口  $i$  到  $j$  的最短路径是相应网络中顶点  $i$  到  $j$  的最短路径。

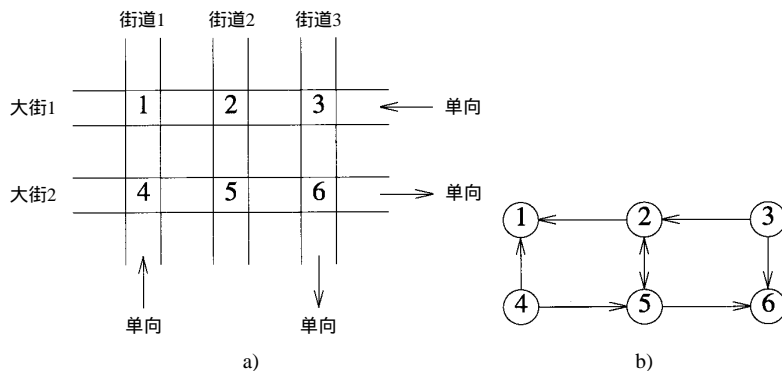


图12-2 街道及其相应的有向图

a) 街道地图 b) 有向图

例12-2 [生成树] 设 $G=(V,E)$ 是一个无向图，当且仅当 $G$ 中每一对顶点之间有一条路径时，可认为 $G$ 是连通的 (connected)。图12-1a 中的无向图是连通的，而 $b$  中的无向图不是。假定 $G$ 是一个通信网络， $V$ 是城市的集合， $E$ 是通信链路的集合。当且仅当 $G$ 是连通的时候， $V$ 中的每一对城市之间可以通信。图12-1a 的通信网络中，城市2和4之间可以通过链路2, 3, 4进行通信，而图12-1b 的网络中，城市2和4不能通信。

假设 $G$ 是连通的， $G$ 中的有些边可能不是必需的，因此即使将它从 $G$ 中去掉， $G$ 仍然可以保持连通。在图12-1a 中，即使将边(2,3)和(1,4)去掉，整个图仍可以保持连通。

图 $H$ 是图 $G$ 的子图 (subgraph) 的充要条件是， $H$ 的顶点和边的集合是 $G$ 的顶点和边的集合的子集。环路 (cycle) 的起始节点与结束节点是同一节点。例如，图12-1a 中，1,2,3,1是一个环路。没有环路的无向连通图是一棵树。一棵包含 $G$ 中所有顶点并且是 $G$ 的子图的树是 $G$ 的生成树 (spanning tree)。图12-1a 的生成树如图12-3所示。

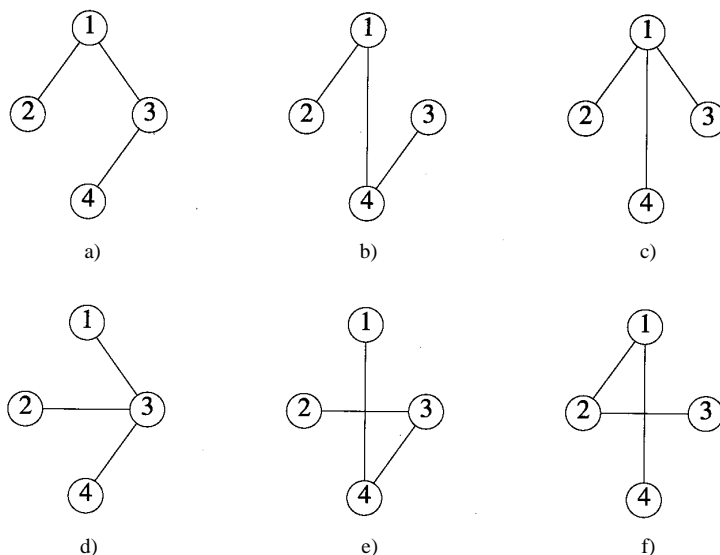


图12-3 图12-1a 的生成树

一个 $n$ 节点的连通图必须至少有 $n-1$ 条边。因此当通信网络的每条链路具有相同的建造费用时，在任意一棵生成树上建设所有的链路可以将网络建设费用减至最小，并且能保证每两个城市之间存在一条通信路径。如果链路具有不同的耗费，那么需要在一棵最小耗费生成树（生成树的耗费是所有边的耗费之和）上建立链路。图 12-4 给出了一个图和它的生成树，图 12-4b 的生成树是一棵最小耗费生成树。

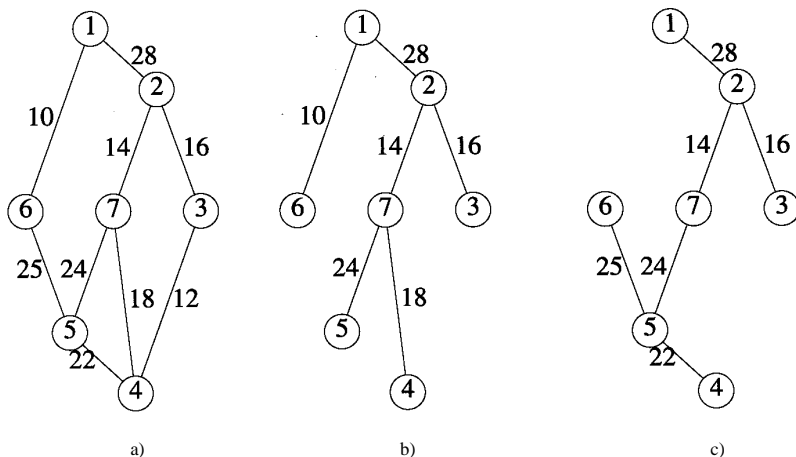


图12-4 连通图和它的两棵生成树

a) 图 b) 耗费为100的生成树 c) 耗费为129的生成树

例12-3 [翻译人员] 假设你正在策划一次国际性会议，此次大会上的所有发言人都只会说英语，而参加会议的其他人说的语言是 $\{L_1, L_2, \dots, L_n\}$ 之一。翻译小组能够将英语与其他语言互译。现在你的任务是如何使翻译小组的人数最少。

可以将这个任务转化为一个图的问题。在这个问题中有两组顶点，一组是相应的翻译人员，一组是语言（如图 12-5 所示）。在翻译人员 $i$ 与语言 $L_j$ 之间存在一条边的充要条件是翻译人员 $i$ 能够将英语和 $L_j$ 互译。当且仅当一条边连接翻译人员和语言时，翻译人员 $i$ 覆盖语言 $L_i$ 。我们需要找到能够覆盖所有语言顶点的最小翻译人员顶点子集。

图 12-5 有一个有趣的特征：可以将顶点集合分成两个子集 $A$ （翻译人员顶点）和 $B$ （语言顶点），这样每条边在 $A$ 中有一个端点，在 $B$ 中有一个端点，具有这种特征的图叫作二分图（bipartite graph）。

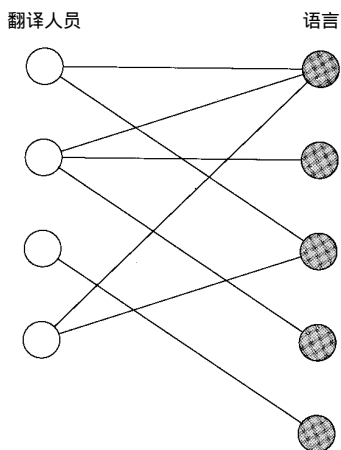


图12-5 翻译人员和语言

## 12.3 特性

设 $G$ 是一个无向图，顶点 $i$ 的度（degree） $d_i$ 是与顶点 $i$ 相连的边的个数。对于图 12-1a， $d_1=3, d_2=2, d_3=3, d_4=2$ 。

特性1 设 $G=(V, E)$ 是一个无向图, 令 $|V|=n, |E|=e, d_i$ 为顶点 $i$ 的度, 则

$$1) \sum_{i=1}^n d_i = 2e$$

$$2) 0 \leq e \leq n(n-1)/2$$

证明 要证明1), 注意到无向图中的每一条边与两个顶点相连, 因此顶点的度之和等于边的数量的2倍。对于2), 一个顶点的度是在0到 $n-1$ 之间, 因此度的和在0到 $n(n-1)$ 之间, 从1)可知,  $e$ 是在0到 $n(n-1)/2$ 之间。

一个具有 $n$ 个顶点,  $n(n-1)/2$ 条边的图是一个完全图 (complete graph)。图12-6给出了 $n=1, 2, 3$ 和4时的完全图。 $K_n$ 代表 $n$ 顶点的完全图。

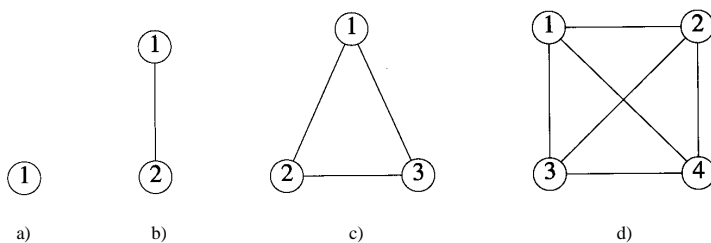


图12-6 完全图

a)  $K_1$  b)  $K_2$  c)  $K_3$  d)  $K_4$

设 $G$ 是一个有向图, 顶点 $i$ 的入度 (in-degree)  $d_i^{in}$ 是指关联至顶点 $i$ 的边的数量。顶点 $i$ 的出度 (out-degree)  $d_i^{out}$ 是指关联于该顶点的边的数量。对于图12-1c的有向图,  $d_1^{in}=0, d_1^{out}=0, d_2^{in}=1, d_2^{out}=1, d_3^{in}=2, d_3^{out}=2$ 。

特性2 设 $G=(V, E)$ 是一个有向图,  $n$ 和 $e$ 的定义与特性1相同, 则

$$1) 0 \leq e \leq n(n-1)$$

$$2) \sum_{i=1}^n d_i^{in} = \sum_{i=1}^n d_i^{out} = e$$

证明 在练习2中, 将要求完成这个特性的证明。

一个 $n$ 顶点的完全有向图 (complete digraph) 包含 $n(n-1)$ 条有向边, 图12-7给出了 $n=1, 2, 3$ 和4时的完全有向图。

入度和出度在无向图中可以作为度的同义词。本节提供的定义可以直接扩充到网络中。

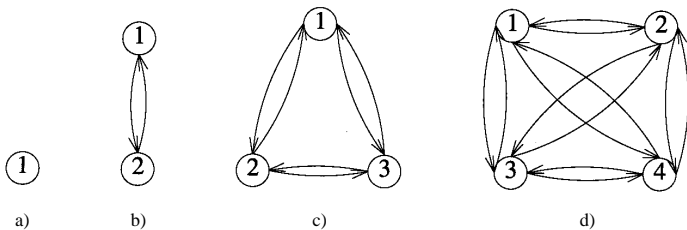


图12-7 完全有向图

a)  $K_1$  b)  $K_2$  c)  $K_3$  d)  $K_4$

## 练习

1. 对于图12-8的每一个有向图, 确定下列各项:

- 1) 每个顶点的入度。
- 2) 每个顶点的出度。
- 3) 邻接于顶点2的顶点集合。
- 4) 邻接至顶点1的顶点集合。
- 5) 关联于顶点3的边的集合。
- 6) 关联至顶点4的边的集合。
- 7) 所有的有向环路和它们的长度。

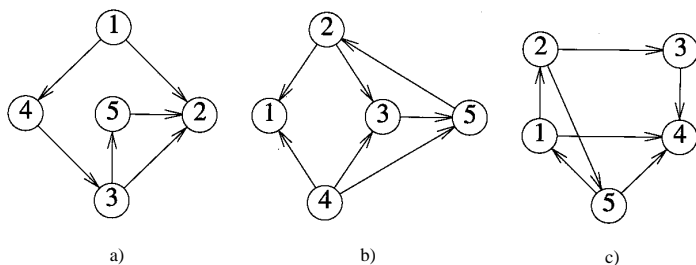


图12-8 有向图

2. 证明特性2。
3. 设 $G$ 是任意无向图，证明有偶数个度数为奇数的顶点。
4. 设 $G=(V,E)$ 是 $|V| > 1$ 的连通图，证明 $G$ 中包含一个度数为1的顶点或一个环路（或两者都有）。
5. 设 $G=(V,E)$ 是至少包含一个环路的连通图，边 $(i, j)$ 至少出现在一个环路中。证明图 $h=(V, E-\{(i, j)\})$ 也是连通的。
6. 证明：
  - 1) 对于每一个 $n (n \geq 1)$ ，都存在一个包含 $n-1$ 条边的无向连通图。
  - 2) 每一个 $n$ 顶点的无向连通图至少有 $n-1$ 条边。可以使用练习4, 5的结论。
7. 一个有向图是强连通（strongly connected）的充要条件是：对于每一对不同顶点 $i$ 和 $j$ ，从 $i$ 到 $j$ 和从 $j$ 到 $i$ 都有一个有向路径。
  - 1) 证明对于每一个 $n (n \geq 2)$ ，都存在一个包含 $n$ 条边的强连通有向图。
  - 2) 证明每一个 $n (n \geq 2)$ 顶点的强连通有向图至少包含 $n$ 条边。
  - 3) 写一个过程确定有向图 $G$ 是否是强连通的。
  - 4) 当 $G$ 是一个邻接矩阵或链接相邻表时，分析程序的时间复杂性。

## 12.4 抽象数据类型 Graph 和 Digraph

抽象数据类型 Graph 专指无向图而抽象数据类型 Digraph 专指有向图。ADT 12-1和12-2的抽象数据类型描述只列出了图操作中的一小部分。在后面的讲述过程中，将不断地增加相应的操作。

抽象数据类型 WeightedGraph 和 WeightedDigraph 相似，只有 Add 操作的描述需要改变以反映与新添加边相关的权值。

### ADT 12-1 无向图的抽象数据类型描述

抽象数据类型 Graph {



实例

顶点集合  $V$  和边集合  $E$

操作

$Create(n)$  : 创建一个具有  $n$  个顶点、没有边的无向图

$Exist(i, j)$  : 如果存在边  $(i, j)$  则返回 true , 否则返回 false

$Edges()$  : 返回图中边的数目

$Vertices()$  : 返回图中顶点的数目

$Add(i, j)$  : 向图中添加边  $(i, j)$

$Delete(i, j)$  : 删除边  $(i, j)$

$Degree(i)$  : 返回顶点  $i$  的度

$InDegree(i)$  : 返回顶点  $i$  的度

$OutDegree(i)$  : 返回顶点  $i$  的度

#### ADT 12-2 有向图的抽象数据类型描述

抽象数据类型  $Graph$  {

实例

顶点集合  $V$  和边集合  $E$

操作

$Create(n)$  : 创建一个具有  $n$  个顶点、没有边的有向图

$Exist(i, j)$  : 如果存在边  $(i, j)$  则返回 true , 否则返回 false

$Edges()$  : 返回图中边的数目

$Vertices()$  : 返回图中顶点的数目

$Add(i, j)$  : 向图中添加边  $(i, j)$

$Delete(i, j)$  : 删除边  $(i, j)$

$Degree(i)$  : 返回顶点  $i$  的度

$InDegree(i)$  : 返回顶点  $i$  的入度

$OutDegree(i)$  : 返回顶点  $i$  的出度

## 练习

8. 请给出加权无向图  $WeightedGraph$  的ADT 描述。

9. 请给出加权有向图  $WeightedDigraph$  的ADT 描述。

## 12.5 无向图和有向图的描述

无向图和有向图最常用的描述方法都是基于邻接的方式：邻接矩阵，邻接压缩表和邻接链表。

### 12.5.1 邻接矩阵

一个  $n$  顶点的图  $G=(V, E)$  的邻接矩阵 ( adjacency matrix ) 是一个  $n \times n$  矩阵  $A$  ,  $A$  中的每一个元素是 0 或 1。假设  $V=\{1, 2, \dots, n\}$ 。如果  $G$  是一个无向图，那么  $A$  中的元素定义如下：

$$A(i, j) = \begin{cases} 1 & \text{如果 } (i, j) \in E \text{ 或 } (j, i) \in E \\ 0 & \text{其它} \end{cases} \quad (12-1)$$

如果G是有向图，那么A中的元素定义如下：

$$A(i, j) = \begin{cases} 1 & \text{如果 } (i, j) \in E \\ 0 & \text{其它} \end{cases} \quad (12-2)$$

图12-1的邻接矩阵如图12-9所示。

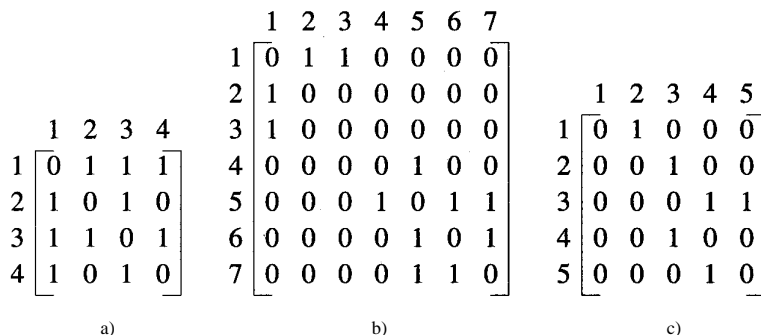


图12-9 图12-1对应的邻接矩阵

从(12-1)和(12-2)中可以得到如下结论：

- 1) 对于 $n$ 顶点的无向图，有 $A(i, i)=0, 1 \leq i \leq n$ 。
- 2) 无向图的邻接矩阵是对称的，即 $A(i, j)=A(j, i), 1 \leq i \leq n, 1 \leq j \leq n$ 。
- 3) 对于 $n$ 顶点的无向图，有 $\sum_{j=1}^n A(i, j) = \sum_{j=1}^n A(j, i) = d_i$  ( $d_i$ 是顶点 $i$ 的度)。
- 4) 对于 $n$ 顶点的有向图，有 $\sum_{j=1}^n A(i, j) = d_i^{out}, \sum_{j=1}^n A(j, i) = d_i^{in}, 1 \leq i \leq n$ 。

#### 1. 将邻接矩阵映射到数组

使用映射 $A(i, j)=a[i][j]$ 可以将 $n \times n$ 的邻接矩阵映射到一个 $(n+1) \times (n+1)$ 的整型数组 $a$ 中。如果 $sizeof(int)$ 等于2个字节，映射的结果需要 $2(n+1)^2$ 字节的存储空间。另一种方法是，采用 $n \times n$ 数组 $a[n][n]$ 和映射 $A(i, j)=a[i-1][j-1]$ 。这种映射需要 $2n^2$ 字节，比前一种减少了 $4n+2$ 个字节。

注意到所有对角线元素都是零而不需要储存，所以还可以进一步减少 $2n$ 字节的存储空间。当把对角线元素去掉后，可得到一个上(或下)三角矩阵(见4.3.3节)。这些矩阵可以被压缩到一个 $(n-1) \times n$ 的矩阵中，如图12-10所示。图中的阴影部分是原邻接矩阵的下三角部分。

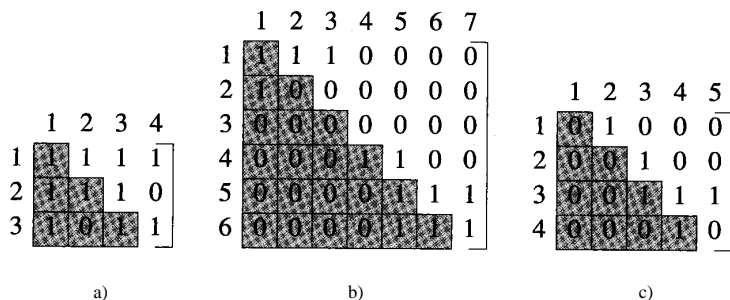


图12-10 图12-9去掉对角线元素后的邻接矩阵

注意到每个邻接矩阵元素只需要1位的存储空间，而每个数组元素需要16位，所以还可以

进一步缩减存储空间。通过使用 unsigned int 类型的数组  $a$ ，可以将  $A$  中的 16 个元素压缩到  $a$  中的 1 个元素中，因此，对存储空间的需求将变为  $n(n-1)/8$  字节。在空间减少的同时，存储和检索邻接矩阵中元素所需要的时间将增加。

对于无向图，邻接矩阵是对称的（见 4.3.5），因此只需要存储上三角（或下三角）的元素，所需空间仅为  $(n^2-n)/2$  位。

## 2. 时间需求

使用邻接矩阵时，需要用  $\Theta(n)$  时间来确定邻接至或邻接于一个给定节点的集合。寻找图中的边数也需要  $\Theta(n)$  的时间。另外，增加或删除一条边需要  $\Theta(1)$  时间。

### 12.5.2 邻接压缩表

在  $G (G=(V,E), |V|=n, |E|=e)$  的邻接压缩表 (packed-adjacency-list) 的定义中，使用了两个一维数组  $h[0:n+1]$  和  $l[0:x]$ ，如果  $G$  是有向图，则  $x=e-1$ ；如果  $G$  是无向图，则  $x=2e-1$ 。首先将所有邻接于顶点 1 的顶点加入到  $l$  中，然后将所有邻接于顶点 2 的顶点加入到  $l$  中，再将邻接于顶点 3 的顶点加入到  $l$  中，这样一直进行下去。（如果  $i$  和  $j$  是无向图的邻接顶点，那么  $i$  邻接于  $j$ ， $j$  邻接于  $i$ ）。下面构造  $h$ ，使得当  $h[i] < h[i+1]$  时，邻接于顶点  $i$  的所有顶点的位置是  $l[h[i]]$ ， $l[h[i]+1]$ ， $\dots$ ， $l[h[i+1]-1]$ ；当  $h[i] = h[i+1]$  时，没有邻接于  $i$  的顶点。则  $l[h[i]]$ ， $l[h[i]+1]$ ， $\dots$ ， $l[h[i+1]-1]$  是顶点  $i$  的邻接压缩表。顶点在表中的次序并不重要。图 12-11 给出了图 12-1 所对应的邻接压缩表。

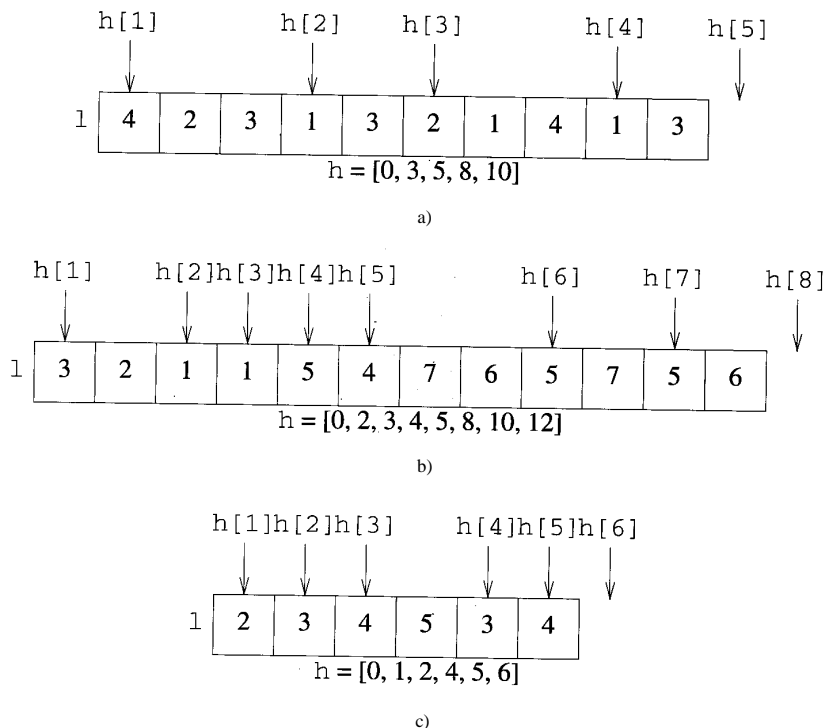


图12-11 图12-1对应的邻接压缩表

对于无向图， $h$  的取值范围是  $0 \sim 2e$ 。因为这个范围只能描述  $2e+1$  个互不相同的值，所以

每一个  $h[i]$  最多需要  $\lceil \log(2e+1) \rceil$  位。数组  $l$  的范围为  $1 \sim n$ ，因此  $l$  的每个元素最多需  $\lceil \log n \rceil$  位。所以，每一个  $n$  顶点  $e$  条边的无向图的邻接压缩表所需要的总的存储空间最多为  $(n+1) \lceil \log(2e+1) \rceil + 2e \lceil \log n \rceil = O((n+e) \log n)$ 。

#### 时间需求

当  $e$  远远小于  $n^2$  时，邻接压缩表需要的空间远远小于邻接矩阵需要的空间。如果  $G$  为无向图，顶点  $i$  的度是  $h[i+1]-h[i]$ ， $G$  中边的数目是  $h[n+1]/2$ 。使用邻接表可以比使用邻接矩阵更容易确定这些数量。增加或删除一条边需要  $O(n+e)$  的时间。

### 12.5.3 邻接链表

在邻接链表 (linked-adjacency-list) 中，邻接表是作为链表保存的，可以用类 `Chain<int>` (见程序 3-8) 来实现。另外，可使用一个 `Chain<int>` 类型的头节点数组  $h$  来跟踪这些邻接表。 $h[i].first$  指向顶点  $i$  的邻接表中的第一个节点。如果  $x$  指向链表  $h[i]$  中的一个节点，那么  $(i, x.data)$  是图中的一条边。图 12-12 给出了一些邻接链表。

假设每个指针和整数均为 2 字节长，则一个  $n$  顶点图的邻接链表所需要的空间为  $2(n+m+1)$ ，其中对于无向图， $m=2e$ ；而对于有向图， $m=e$ 。 $l$  可从大小为  $n+2$  的数组  $h$  中得到，如果用  $h[i-1]$  指向顶点  $i$  的链表，可以不需要  $l$ 。

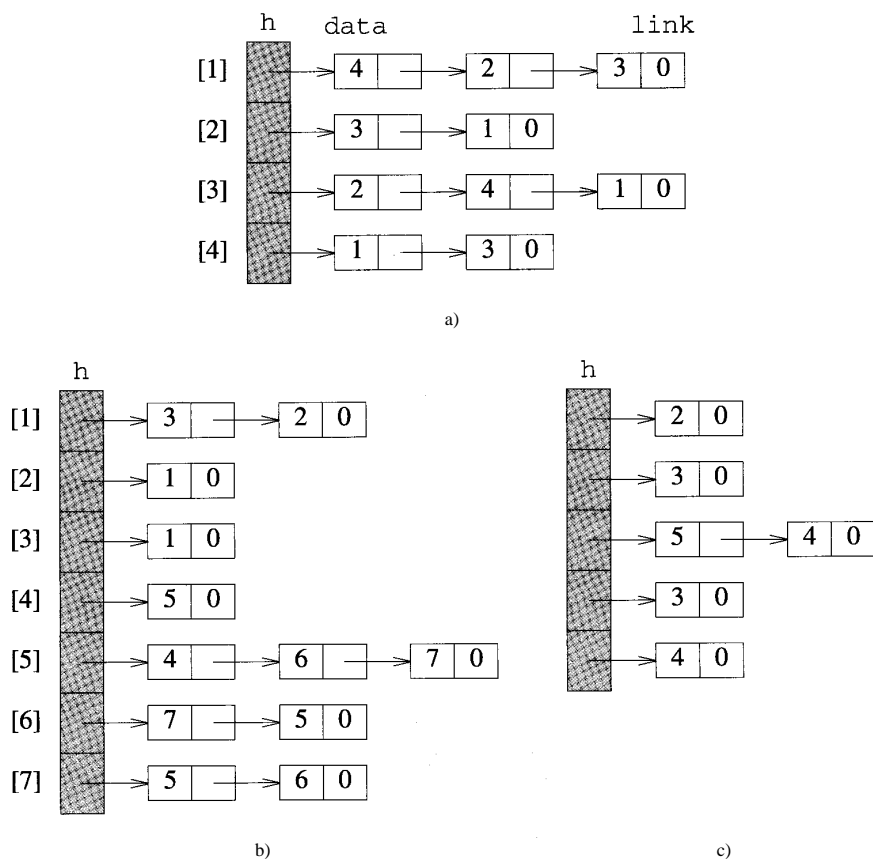


图12-12 图12-1对应的邻接链表

时间需求

邻接链表便于进行边的插入和删除操作。确定邻接表中顶点的数目所需要的时间与表中顶点的数目成正比。

## 练习

10. 请为图12-5和12-8a 提供下列描述：

- 1) 邻接矩阵。
- 2) 邻接压缩表。
- 3) 邻接链表。

11.  $a$  是一个  $(n-1) \times n$  的数组，用来描述一个  $n$  顶点图的邻接矩阵  $A$ 。 $a$  中没有描述矩阵的对角线（如图12-10所示）。编写两个函数 Store 和 Retrieve 分别存储和搜索  $A(i, j)$  的值，每个函数的复杂性应为  $\Theta(1)$ 。

12. 用全邻接矩阵（见图12-9）完成练习11，矩阵的16个元素可压缩成数组  $a$  的一个元素，其中  $a$  是 unsigned int 类型的一维数组。

13. 用无向图完成练习11，仅在一维数组  $a$  中存储无向图的下三角矩阵。假设  $a$  中的每个元素非0即1。

14. 用无向图完成练习11，无向图的下三角矩阵存储在一个 unsigned int 类型的一维数组  $a$  中，假设  $a$  中的每一个元素描述下三角矩阵中的16个元素。

15. 假设用一个  $n \times n$  的数组  $a$  来描述一个有向图的  $n \times n$  邻接矩阵

- 1) 编写一个函数确定一个顶点的出度，函数的复杂性应为  $\Theta(n)$ 。
- 2) 编写一个函数确定一个顶点的入度，函数的复杂性应为  $\Theta(n)$ 。
- 3) 编写一个函数确定图中边的数目，函数的复杂性应为  $\Theta(n^2)$ 。

16. 假设用邻接压缩表描述一个无向图

- 1) 编写一个函数删除边  $(i, j)$ 。代码的复杂性是多少？
- 2) 编写一个函数增加边  $(i, j)$ 。代码的复杂性是多少？

17. 对有向图完成练习16。

18. 用邻接链表完成练习15。

19. 用邻接链表完成练习16。

20.  $G$  是一个  $n$  顶点， $e$  条边的无向图。 $e$  至少是多少时， $G$  的邻接矩阵所占用的空间才会比邻接压缩表所占用的空间少？

21. 对有向图  $G$  完成练习20。

## 12.6 网络描述

将图和有向图的描述进行简单扩充就可得到网络的描述，无论它是加权的无向图还是有向图。类似于邻接矩阵的描述，可用一个矩阵  $C$  来描述耗费邻接矩阵（cost-adjacency-matrix）。如果  $A(i, j)$  是1，那么  $C(i, j)$  是相应边的耗费（或权）；如果  $A(i, j)$  是0，那么相应的边不存在， $C(i, j)$  等于某些预置的值 NoEdge。选择 NoEdge 是为了便于区分边是否存在。一般来说，NoEdge 的值被设为无穷大。图12-13给出了图12-1的耗费邻接矩阵。符号  $\infty$  代表 NoEdge 的值。

用（顶点，权）替换  $l$  中的每一个入口，可以从相应的无权图或无权有向图中得到网络的邻接压缩表。例如，相应于图12-1c 的数组  $l$  如图12-11c 所示，每条边的权值由图12-13c 中的耗

费邻接矩阵给出。相应的加权有向图的数组  $l$  为  $[(2,8), (3,3), (4,2), (5,7), (3,6), (4,5)]$ 。数组  $h$  不变。

	1	2	3	4	5	6	7
1	$\infty$	9	5	$\infty$	$\infty$	$\infty$	$\infty$
2	9	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
3	5	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
4	$\infty$	$\infty$	$\infty$	$\infty$	3	$\infty$	$\infty$
5	$\infty$	$\infty$	$\infty$	3	$\infty$	6	4
6	$\infty$	$\infty$	$\infty$	$\infty$	6	$\infty$	1
7	$\infty$	$\infty$	$\infty$	$\infty$	4	1	$\infty$

	1	2	3	4
1	$\infty$	8	$\infty$	$\infty$
2	$\infty$	$\infty$	3	$\infty$
3	$\infty$	$\infty$	$\infty$	2
4	$\infty$	$\infty$	6	$\infty$
5	$\infty$	$\infty$	$\infty$	5

a)

b)

c)

图12-13 图12-1对应的可能的耗费邻接矩阵

使用 `Chain<GraphNode>` 类型的链表，可以从相应的图的邻接表描述中得到网络的邻接表描述，其中 `GraphNode` 包括两个部分：`vertex` 和 `weight`。图12-14给出了与图12-13a 的耗费邻接矩阵相对应的网络描述。图中每个节点的第一部分是顶点，第二部分是权。

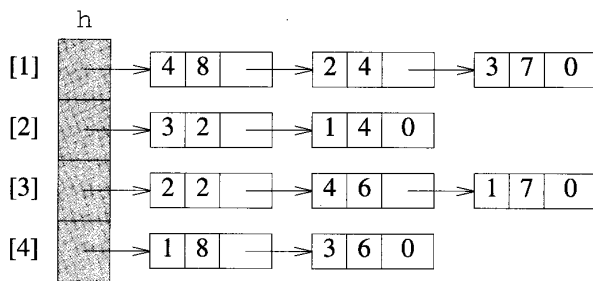


图12-14 图12-13a对应的网络的邻接链表

## 练习

22. 给出相应于图12-1a 和b 中耗费邻接矩阵的网络邻接压缩表。
23. 给出相应于图12-1a 和b 中耗费邻接矩阵的网络邻接链表。

## 12.7 类定义

### 12.7.1 不同的类

无权有向图和无向图可以看作每条边的权是 1 的加权有向图和无向图。因此 12.4 节中定义的抽象数据类型 (`Graph`, `Digraph`, `WeightedGraph`, `WeightedDigraph`) 是一个更普通的抽象数据类型 `Network` 的子类。

对于 12.4 节的四种抽象数据类型中的每一种，考虑 12.5 和 12.6 节所讨论的三种描述方法。用 C++ 类把抽象数据类型和描述方法联系起来，可以得到 12 个类。本节只给出其中的八个类，其余四个对应于压缩描述类留作练习（练习 33 至 36）。

本节要讨论的八个类是 `AdjacencyGraph`, `AdjacencyWGraph` (矩阵描述的加权图), `AdjacencyDigraph`, `AdjacencyWDigraph`, `LinkedGraph`, `LinkedWGraph`, `LinkedDigraph` 和 `LinkedWDigraph`。

4 种抽象数据类型中的若干对类型之间存在 IsA 关系，例如，无向图可以看作边  $(i, j)$  和边  $(j, i)$  都存在的有向图；也可以看作所有边的权均为 1 的加权图；或者看作所有边的权为 1，若边  $(i,$

$j$ ) 存在, 则边  $(j, i)$  也存在的加权有向图。类似地, 有向图也可以看作所有边的权均为 1 的加权有向图。

利用这些关系可以很容易地设计这八个类, 因为可以从其中的一个类派生出另一个类。虽然存在很多 IsA 关系, 但只能利用其中少数几个关系。很自然地, 可以从一个邻接矩阵类派生另一个邻接矩阵类, 从一个链接类派生另一个链接类。图 12-15 中的有向无环图给出了各个类之间的派生层次。例如, AdjacencyGraph 类可以由 AdjacencyWGraph 派生而来。对于链接类, 引入另外一个类 LinkBase 来描述链表数组。利用类 LinkBase 可以避免这四个链接类中公用函数的重复。而对于邻接类, 不需要额外定义这样的类, 因为邻接类有一个共同的根类——AdjacencyWDigraph, 在这个根类中定义了所有类的公用函数。

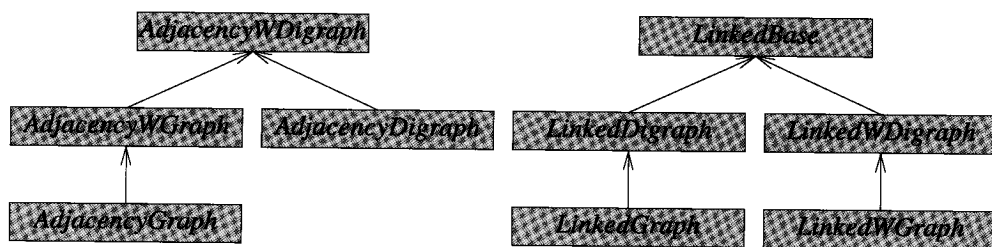


图12-15 类的派生层次

### 12.7.2 邻接矩阵类

邻接矩阵类的根是 AdjacencyWDigraph, 因此从这个类开始。程序 12-1 给出了类的描述。程序中, 先用程序 1-13 中函数 Make2DArray 为二维数组  $a$  分配空间, 然后对数组  $a$  初始化, 以描述一个  $n$  顶点、没有边的图的邻接矩阵, 其复杂性为  $\Theta(n^2)$ 。该代码没有捕获可能由 Make2DArray 引发的异常。在析构函数中调用了程序 1-14 中的二维数组释放函数 Delete2DArray。

程序 12-1 加权有向图的耗费邻接矩阵

```
template<class T>
class AdjacencyWDigraph {
    friend AdjacencyWGraph<T>;
public:
    AdjacencyWDigraph (int Vertices = 10, T noEdge = 0);
    ~AdjacencyWDigraph() {Delete2DArray(a,n+1);}
    bool Exist(int i, int j) const;
    int Edges() const {return e;}
    int Vertices() const {return n;}
    AdjacencyWDigraph<T>& Add (int i, int j, const T& w);
    AdjacencyWDigraph<T>& Delete(int i, int j);
    int OutDegree(int i) const;
    int InDegree(int i) const;
private:
    T NoEdge; // 用于没有边存在的情形
    int n;    // 顶点数目
    int e;    // 边数
```



```

    T **a;    // 二维数组
};

template<class T>
AdjacencyWDigraph<T>::AdjacencyWDigraph(int Vertices, T noEdge)
{// 构造函数
    n = Vertices;
    e = 0;
    NoEdge = noEdge;
    Make2DArray(a, n+1, n+1);
    //初始化为没有边的图
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            a[i][j] = NoEdge;
}

template<class T>
bool AdjacencyWDigraph<T>::Exist(int i, int j) const
{// 边(i, j)存在吗?
    if (i < 1 || j < 1 || i > n || j > n || a[i][j] == NoEdge) return false;
    return true;
}

template<class T>
AdjacencyWDigraph<T>& AdjacencyWDigraph<T> ::Add(int i, int j, const T& w)
{// 如果边 (i,j) 不存在, 则将该边加入有向图中
    if (i < 1 || j < 1 || i > n ||
        j > n || i == j || a[i][j] != NoEdge)
        throw BadInput();
    a[i][j] = w;
    e++;
    return *this;
}

template<class T>
AdjacencyWDigraph<T>& AdjacencyWDigraph<T> ::Delete(int i, int j)
{//删除边(i,j).
    if (i < 1 || j < 1 || i > n || j > n || a[i][j] == NoEdge)
        throw BadInput();
    a[i][j] = NoEdge;
    e--;
    return *this;
}

template<class T>
int AdjacencyWDigraph<T>::OutDegree(int i) const
{// 返回顶点 i的出度

```

```

    if (i < 1 || i > n) throw BadInput();
    // 计算顶点 i 的出度
    int sum = 0;
    for (int j = 1; j <= n; j++)
        if (a[i][j] != NoEdge) sum++;
    return sum;
}

template<class T>
int AdjacencyWDigraph<T>::InDegree(int i) const
{// 返回顶点 i 的入度
    if (i < 1 || i > n) throw BadInput();
    // 计算顶点 i 的入度
    int sum = 0;
    for (int j = 1; j <= n; j++)
        if (a[j][i] != NoEdge) sum++;
    return sum;
}

```

函数Exist 的代码不能区分下面两种情况：1) i 和 / 或j 是否为有效顶点；2) 边(i, j) 是否存在。可以对代码进行修改，使前一种情况引发一个异常 OutOfBounds。对于Add和Delete函数也可以如法炮制。所有代码简单易懂，所以将不再作进一步说明。Exist, Edges, Add和Delete的复杂性均为  $\Theta(1)$ ，而OutDegree和InDegree的复杂性为  $\Theta(n)$ 。

剩下的三种邻接矩阵类在程序12-2到程序12-4中给出。

程序12-2 加权图的耗费邻接矩阵

```

template<class T>
class AdjacencyWGraph : public AdjacencyWDigraph<T> {
public:
    AdjacencyWGraph(int Vertices = 10, T noEdge = 0) : AdjacencyWDigraph<T>(Vertices, noEdge) {}
    AdjacencyWGraph<T>& Add(int i, int j, const T& w)
    {AdjacencyWDigraph<T>::Add(i, j, w);
     a[j][i] = w;
     return *this;}
    AdjacencyWGraph<T>& Delete(int i, int j)
    {AdjacencyWDigraph<T>::Delete(i, j);
     a[j][i] = NoEdge;
     return *this;}
    int Degree(int i) const {return OutDegree(i);}
};

```

程序12-3 有向图的邻接矩阵

```

class AdjacencyDigraph : public AdjacencyWDigraph<int> {
public:
    AdjacencyDigraph(int Vertices = 10) : AdjacencyWDigraph<int>(Vertices, 0) {}
};

```

```
AdjacencyDigraph& Add(int i, int j)
{AdjacencyWDigraph<int>::Add(i,j,1);
 return *this;}
AdjacencyDigraph& Delete(int i, int j)
{AdjacencyWDigraph<int>::Delete(i,j);
 return *this;}
};
```

程序12-4 图的邻接矩阵

```
class AdjacencyGraph : public AdjacencyWGraph<int>
{
public:
    AdjacencyGraph(int Vertices = 10) : AdjacencyWGraph<int>(Vertices, 0) {}
    AdjacencyGraph& Add(int i, int j)
    {AdjacencyWGraph<int>::Add(i,j,1);
     return *this;}
    AdjacencyGraph& Delete(int i, int j)
    {AdjacencyWGraph<int>::Delete(i,j);
     return *this;}
};
```

### 12.7.3 扩充Chain类

在链表描述中，对象被描述为一个链表数组，且每个链表的类型为 Chain类（见程序3-8）。我们所需要的一种链表操作目前尚未定义，因此下面增加该操作。

新的共享成员函数（见程序12-5）删除一个具有指定关键值的元素。程序在链表中搜索与x的关键值相同的元素（假设操作符!=已被重载用于比较两个元素的关键值）。如果找到了匹配的元素，将它从链表中删除，并返回到x中。

程序12-5 从链表中删除元素

```
template<class T>
Chain<T>& Chain<T>::Delete(T& x)
{// 删除与 x匹配的元素
// 如果不存在相匹配的元素，则引发异常 BadInput
    ChainNode<T> *current = first,
        *trail = 0; // 指向current之后的节点

    //搜索匹配元素
    while (current && current->data != x) {
        trail = current;
        current = current->link;}
    if (!current) throw BadInput(); // 不存在匹配元素

    //在节点current中找到匹配元素
```

```

x = current->data; // 保存匹配元素

// 从链表中删除 current 节点
if (trail) trail->link = current->link;
else first = current->link;

delete current; // 释放节点
return *this;
}

```

#### 12.7.4 类LinkedBase

如图12-15所示，无权图和加权图的派生路径之所以不同，其原因在于加权有向图和无向图的链表节点中有一个权值域，而无权有向图和无向图中则没有。对于后者，使用 `int` 类型的链节点就足够了；而对于前者，链节点必须包含一个权值域和一个顶点域。尽管节点结构存在这种差别，但某些基本函数的代码仍然是一样的。因此，引入一个新类 `LinkedBase`（见程序12-6），它包含了构造函数、析构函数、`Edges`和`OutDegree`函数。

构造函数为链表数组分配空间。`h[i]` 是顶点  $i$  的链表， $1 \leq i \leq n$ 。析构函数释放这些空间。构造函数、析构函数及 `Edges` 的复杂性均为  $\Theta(1)$ ，`OutDegree( $i$ )` 的复杂性为  $\Theta(d_i^{out})$ 。

程序12-6 邻接链描述的基类

```

template<class T>
class LinkedBase {
    friend class LinkedDigraph;
    friend class LinkedGraph;
    friend LinkedWDigraph<int>;
    friend LinkedWGraph<int>;
public:
    LinkedBase(int Vertices = 10)
    {n = Vertices;
     e = 0;
     h = new Chain<T> [n+1];}
    ~LinkedBase() {delete [] h;}
    int Edges() const {return e;}
    int Vertices() const {return n;}
    int OutDegree(int i) const
    {if (i < 1 || i > n) throw OutOfBounds();
     return h[i].Length();}
private:
    int n;    //顶点数
    int e;    // 边数
    Chain<T> *h; // 邻接矩阵
};

```

## 12.7.5 链接类

前面所定义的四类链接类是 `LinkBase` 类的友元。程序 12-7 给出了 `LinkDigraph` 类。程序中增加了一个保护成员函数 `AddNoCheck`，在添加一条边时该函数不作任何检查。之所以增加这个函数是因为在使用邻接表时，有效性检查的开销很大，既然能够知道所增加的边是有效的，所以省略了这种检查。`Exist(i, j)` 和 `Add(i, j)` 的复杂性为  $\Theta(d_i^{out})$ ，而 `AddNoCheck` 的复杂性为  $\Theta(1)$ ，`Delete(i, j)` 的复杂性为  $\Theta(d_i^{out} + d_j^{out})$ ，`InDegree` 的复杂性为  $\Theta(n+e)$ 。

程序 12-7 有向图的邻接链表

```
class LinkDigraph : public LinkBase<int> {
public:
    LinkDigraph(int Vertices = 10) : LinkBase<int>(Vertices) {}
    bool Exist(int i, int j) const;
    LinkDigraph& Add(int i, int j);
    LinkDigraph& Delete(int i, int j);
    int InDegree(int i) const;
protected:
    LinkDigraph& AddNoCheck(int i, int j);
};

bool LinkDigraph::Exist(int i, int j) const
{// 边 (i,j) 存在吗?
    if (i < 1 || i > n) throw OutOfBounds();
    return (h[i].Search(j)) ? true : false;
}

LinkDigraph& LinkDigraph::Add(int i, int j)
{// 把边 (i,j) 加入到图中
    if (i < 1 || j < 1 || i > n || j > n || i == j || Exist(i, j)) throw BadInput();
    return AddNoCheck(i, j);
}

LinkDigraph& LinkDigraph::AddNoCheck(int i, int j)
{// 增加边但不检查可能出现的错误
    h[i].Insert(0, j); // 把 j 添加到顶点 i 的表中
    e++;
    return *this;
}

LinkDigraph& LinkDigraph::Delete(int i, int j)
{// 删除边 (i,j)
    if (i < 1 || i > n) throw OutOfBounds();
    h[i].Delete(j);
    e--;
    return *this;
}
```

```
int LinkedDigraph::InDegree(int i) const
{// 返回顶点 i的入度
    if (i < 1 || i > n) throw OutOfBounds();
    // 计算到达顶点 i的边
    int sum = 0;
    for (int j = 1; j <= n; j++)
        if (h[j].Search(i)) sum++;
    return sum;
}
```

程序12-8给出了类LinkedGraph，它是从类LinkedDigraph派生而来的。除函数InDegree外，其他所有函数的复杂性均与LinkedDigraph类中对应函数的复杂性相同。

程序12-8 LinkedGraph类

```
class LinkedGraph : public LinkedDigraph {
public:
    LinkedGraph(int Vertices = 10) : LinkedDigraph (Vertices) {}
    LinkedGraph& Add(int i, int j);
    LinkedGraph& Delete(int i, int j);
    int Degree(int i) const {return InDegree(i);}
    int OutDegree(int i) const {return InDegree(i);}
protected:
    LinkedGraph& AddNoCheck(int i, int j);
};

LinkedGraph& LinkedGraph::Add(int i, int j)
{// 向图中添加边 (i,j)
    if (i < 1 || j < 1 || i > n || j > n || i == j || Exist(i, j)) throw BadInput();
    return AddNoCheck(i, j);
}

LinkedGraph& LinkedGraph::AddNoCheck(int i, int j)
{// 添加边(i,j), 不检查可能出现的错误
    h[i].Insert(0,j);
    try {h[j].Insert(0,i);}
    // 若出现异常, 则取消第一次插入, 并引发同样的异常
    catch (...) {h[i].Delete(j); throw;}
    e++;
    return *this;
}

LinkedGraph& LinkedGraph::Delete(int i, int j)
{//删除边(i,j)
    LinkedDigraph::Delete(i,j);
    e++; // 补偿
    LinkedDigraph::Delete(j,i);
}
```

```
return *this;
}
```

程序12-9给出了加权有向图的类定义。GraphNode类如程序12-10所示。InDegree的代码与LinkedDigraph类的代码相同，在程序12-9中不再给出。所有函数的复杂性与LinkedDigraph类对应函数的复杂性相同。现在，从LinkedWDigraph类可以派生出LinkedWGraph类(见练习32)。

程序12-9 加权有向图的邻接链表

```
template<class T>
class LinkedWDigraph : public LinkedBase<GraphNode<T> > {
public:
    LinkedWDigraph(int Vertices = 10) : LinkedBase<GraphNode<T> > (Vertices) {}
    bool Exist(int i, int j) const;
    LinkedWDigraph<T>& Add(int i, int j, const T& w);
    LinkedWDigraph<T>& Delete(int i, int j);
    int InDegree(int i) const;
protected:
    LinkedWDigraph<T>&
        AddNoCheck(int i, int j, const T& w);
};
```

```
template<class T>
bool LinkedWDigraph<T>::Exist(int i, int j) const
// 存在边(i,j) 吗?
{
    if (i < 1 || i > n) throw OutOfBounds();
    GraphNode<T> x;
    x.vertex = j;
    return h[i].Search(x);
}
```

```
template<class T>
LinkedWDigraph<T>& LinkedWDigraph<T> ::Add(int i, int j, const T& w)
// 添加边(i,j)
{
    if (i < 1 || j < 1 || i > n || j > n || i == j
        || Exist(i, j)) throw BadInput();
    return AddNoCheck(i, j, w);
}
```

```
template<class T>
LinkedWDigraph<T>& LinkedWDigraph<T> ::AddNoCheck(int i, int j, const T& w)
// 添加边(i,j)，不检查可能出现的错误
{
    GraphNode<T> x;
    x.vertex = j; x.weight = w;
    h[i].Insert(0,x);
    e++;
    return *this;
}
```



```

}

template<class T>
LinkedWDigraph<T>& LinkedWDigraph<T> ::Delete(int i, int j)
{// 删除边(i,j)
    if (i < 1 || i > n) throw OutOfBounds();
    GraphNode<T> x;
    x.vertex = j;
    h[i].Delete(x);
    e--;
    return *this;
}

template<class T>
int LinkedWDigraph<T>::InDegree(int i) const
{// 返回顶点i的入度
    if (i < 1 || i > n) throw OutOfBounds();
    int sum = 0;
    GraphNode<T> x;
    x.vertex = i;
    // 检查所有的(j,i)
    for (int j = 1; j <= n; j++)
        if (h[j].Search(x)) sum++;
    return sum;
}

```

程序12-10 GraphNode类

```

template <class T>
class GraphNode {
    friend LinkedWDigraph<T>;
    friend LinkedWGraph<T>;
    friend Chain<T>;
public:
    int operator !=(GraphNode<T> y) const
    {return (vertex != y.vertex);}
    void Output(ostream& out) const
    {out << vertex << " " << weight << " ";}
private:
    int vertex; // 边的第二个顶点
    T weight; // 边的权重
};

template <class T>
ostream& operator<<(ostream& out, GraphNode<T> x)
    {x.Output(out); return out;}

```

## 练习

24. 编写一个输入无向图的函数 `AdjacencyGraph::Input` 和一个输出函数 `Output`。假设输入内容包括顶点、边的数量以及边的集合。每条边由一对顶点给出。重载操作符 `<<` 以便于输入无向图。

25. 编写一个输入有向图的函数 `AdjacencyDigraph::Input` 和一个输出函数 `Output`，重载操作符 `<<` 以便于输入有向图。

26. 编写一个输入无向网络的函数 `AdjacencyWGraph::Input` 和一个输出函数 `Output`，重载操作符 `<<` 和 `>>`。

27. 编写一个输入有向网络的函数 `AdjacencyWDigraph::Input` 和一个输出函数 `Output`，重载操作符 `<<` 和 `>>`。

28. 编写一个输入无向图的函数 `LinkedGraph::Input` 和一个输出函数 `Output`，重载操作符 `<<` 和 `>>`。

29. 编写一个输入有向图的函数 `LinkedDigraph::Input` 和一个输出函数 `Output`，重载操作符 `<<` 和 `>>`。

30. 编写一个输入无向网络的函数 `LinkedWGraph::Input` 和一个输出函数 `Output`，重载操作符 `<<` 和 `>>`。

31. 编写一个输入有向网络的函数 `LinkedWDigraph::Input` 和一个输出函数 `Output`，重载操作符 `<<` 和 `>>`。

32. 设计一个 C++ 类 `LinkedWGraph`，用邻接链表描述加权无向图，从 `LinkedWDigraph` 类（见程序 12-9）中派生此类。

33. 设计一个 C++ 类 `PackedAdjGraph`，用邻接压缩表描述无向图，从 `LinearList` 类（见程序 3-1）中派生此类。

34. 设计一个 C++ 类 `PackedAdjWGraph`，用邻接压缩表描述加权无向图，从 `LinearList` 类（见程序 3-1）中派生此类。

35. 设计一个 C++ 类 `PackedAdjDigraph`，用邻接压缩表描述有向图，从 `LinearList` 类（见程序 3-1）中派生此类。

36. 设计一个 C++ 类 `PackedAdjWDigraph`，用邻接压缩表描述加权有向图，从 `LinearList` 类（见程序 3-1）中派生此类。

## 12.8 图的遍历

### 12.8.1 基本概念

不论采用哪一种图类编写应用程序，都需要沿着矩阵的一行或多行向下移动，或者沿着一个或多个链表向下移动，实现这种移动的函数称为遍历器（iterator）。对于类 `Chain`，定义了一个附加类 `ChainIterator`（见程序 3-18），它提供了从链表的一个元素移到下一个元素的函数。可以在图类中引入相同策略并定义一些新的、能够提供遍历函数的图类。

不过在图类中，遍历器被嵌入到了类中。在练习 37，38 和 39 中要求设计新的遍历器类。下面将给出遍历函数及其相应说明，这些函数只使用一个游标来跟踪矩阵的每一行或每个链表，因此，它们不支持需要多个游标的应用。

- `Begin(i)` 对于邻接表，返回顶点 `i` 所对应表中的第一个顶点；对于邻接矩阵，返回邻接

于顶点*i*的最小（即第一个）顶点。在两种情况中，如果没有邻接顶点，都将返回零值。

- `NextVertex(i)` 返回顶点*i*对应邻接表中的下一个顶点或返回邻接于顶点*i*的下一个最小顶点。同样，当没有下一个顶点时函数返回零。

- `InitializePos()` 初始化用来跟踪每一个邻接表或(耗费)邻接矩阵每一行中当前位置的存储配置。

- `DeactivatePos()` 取消`InitializePos()`所产生的存储配置。

## 12.8.2 邻接矩阵的遍历函数

邻接矩阵的遍历函数可以作为 `AdjacencyWDigraph`类的一个共享函数来加以实现。由于其他3种邻接类都是从这个类派生出来的，因此它们可以从 `AdjacencyDigraph`类中继承该函数。

由于可能处在邻接矩阵不同行的不同位置，因此用一个数组 `pos`来记录每一行中的位置，这个变量是 `AdjacencyWDigraph`的私有成员，定义如下：

```
int *pos;
```

遍历函数的代码见程序 12-11。

程序 12-11 邻接矩阵的遍历函数

---

```
void InitializePos() {pos = new int [n+1];}

void DeactivatePos() {delete [] pos;}

template<class T>
int AdjacencyWDigraph<T>::Begin(int i)
{//返回第一个与顶点 i邻接的顶点
if (i < 1 || i > n) throw OutOfBounds();

// 查找第一个邻接顶点
for (int j = 1; j <= n; j++)
    if (a[i][j] != NoEdge) {j 是第一个
        pos[i] = j;
        return j;}

pos[i] = n + 1; // 没有邻接顶点
return 0;
}

template<class T>
int AdjacencyWDigraph<T>::NextVertex(int i)
{// 返回下一个与顶点 i邻接的顶点
if (i < 1 || i > n) throw OutOfBounds();

// 寻找下一个邻接顶点
for (int j = pos[i] + 1; j <= n; j++)
    if (a[i][j] != NoEdge) {j 是下一个顶点
        pos[i] = j; return j;}

pos[i] = n + 1; // 不存在下一个顶点
```

```
    return 0;
}
```

### 12.8.3 邻接链表的遍历函数

对于用邻接链表描述的图和网络，需要将程序 12-12中定义的共享成员函数 Initialize和 DeactivatePos加入到LinkedBase类中，并且，还需要定义一个私有变量 pos：

```
ChainIterator<T> *pos;
```

此外，还需要将余下的两个遍历函数加入到 LinkedDigraph和LinkedWDigraph类中，代码见程序12-13和12-14。

程序12-12 加入到LinkedBase类中

```
void InitializePos()
{pos = new ChainIterator<T> [n+1];}
void DeactivatePos() {delete [] pos;}
```

程序12-13 邻接链表的遍历函数

```
int LinkedDigraph::Begin(int i)
{// 返回第一个与顶点 i邻接的顶点
    if (i < 1 || i > n) throw OutOfBounds();
    int *x = pos[i].Initialize(h[i]);
    return (x) ? *x : 0;
}
```

```
int LinkedDigraph::NextVertex(int i)
{// 返回下一个与顶点 i邻接的顶点
    if (i < 1 || i > n) throw OutOfBounds();
    int *x = pos[i].Next();
    return (x) ? *x : 0;
}
```

程序12-14 链接加权有向图的遍历函数

```
template<class T>
int LinkedWDigraph<T>::Begin(int i)
{// 返回第一个与顶点 i邻接的顶点
    if (i < 1 || i > n) throw OutOfBounds();
    GraphNode<T> *x = pos[i].Initialize(h[i]);
    return (x) ? x->vertex : 0;
}
```

```
template<class T>
int LinkedWDigraph<T>::NextVertex(int i)
{// 返回下一个与顶点 i邻接的顶点
    if (i < 1 || i > n) throw OutOfBounds();
    GraphNode<T> *x = pos[i].Next();
```

```
return (x) ? x->vertex : 0;  
}
```

## 练习

37. 为AdjacencyWDigraph 设计一个遍历器类，并加以测试。类应提供本节中所介绍的各种遍历功能。

38. 用LinkedDigraph 完成练习37。

39. 用LinkedWDigraph 完成练习37。

## 12.9 语言特性

### 12.9.1 虚函数和多态性

考察LinkedGraph::Add (见程序12-8) 和LinkedDigraph::Add (见程序12-7) 的代码，两段程序是一样的！由于LinkedGraph是从LinkedDigraph 派生而来，尝试一下将LinkedGraph::Add 删除并从 LinkedDigraph 中继承 Add成员函数。假设继承了这个成员，如果 G的类型是LinkedGraph，那么表达式G.Add(i,j)将调用所继承的函数 LinkedDigraph::Add，该函数又将调用函数LinkedDigraph::AddNoCheck，并将边加入到链表i 而不是链表j 中。LinkedDigraph::Add的行为是单态的 (unimorphic)，也就是说，不管LinkedDigraph::Add是作用于LinkedDigraph类的对象还是作用于LinkedDigraph派生类的对象，LinkedDigraph::Add的行为都是一样的，所调用的函数也完全相同。

由于LinkedDigraph有它自己的AddNoCheck函数，因此，当 LinkedDigraph::Add作用于LinkedGraph类的对象上时，希望它调用LinkedGraph::AddNoCheck；当它作用于 LinkedDigraph类的对象上时，希望它调用LinkedDigraph::AddNoCheck。即，希望LinkedDigraph::Add的行为是多态的 (polymorphic)，被调用的函数取决于函数所作用的对象类型。通过在程序 12-7的第10行：

```
LinkedDigraph& AddNoCheck( int i , int j ) ;
```

之前加上关键字virtual，就能把LinkedDigraph::AddNoCheck变成一个虚函数 (virtual function)。此外不需要其他任何修改，特别不需要在以下函数之前加入关键字 virtual：

```
LinkedDigraph& LinkedDigraph::AddNoCheck(int i, int j)
```

虚函数可用一种特殊的方式来处理。首先，考虑单一继承的情况，在此情况中，类既可以是基类，也可是另一个类的派生类。假设 A是B的一个派生类且A和B都至少包含一个虚函数。可为A构造一个虚函数表，对于A和B的每一个虚函数F，表中都有一个对应的指针，用来指向调用A.F时实际执行的函数F。

考察图12-16的派生结构，这是一个单一继承的例子，因为每一个类最多是从一个类派生而来。这里有4个类A,B,C和D。类A从类B派生而来，类B从类C派生而来，而类C又从类D派生而来。方框中列出了类的成员函数。例如，类D包含虚函数f和g（在图中，vf是virtual f的缩写，vg是virtual g的缩写）和非虚函数h。函数D::f和D::g仅输出字符D，而A::f和A::g输出字符A。虽然g不是B中显式声明的虚函数，它仍然是一个虚函数，因为在D中它是虚函数。一旦一个函数被声明为虚函数，它在所有的派生类中仍然是虚函数。

图12-16中方框的右边给出了类的虚函数表。表中包含一个指针，指向每个虚函数的实际

执行函数。由于D不是派生类，它的表中只包含D中所定义的虚函数。其他每个类的虚函数表可从其父类的虚函数表中构造出来。例如，C的虚函数表中包含C中新定义的虚函数以及对D中的虚函数进行修改后的函数。可采用同样的方法来构造A和B的虚函数表。

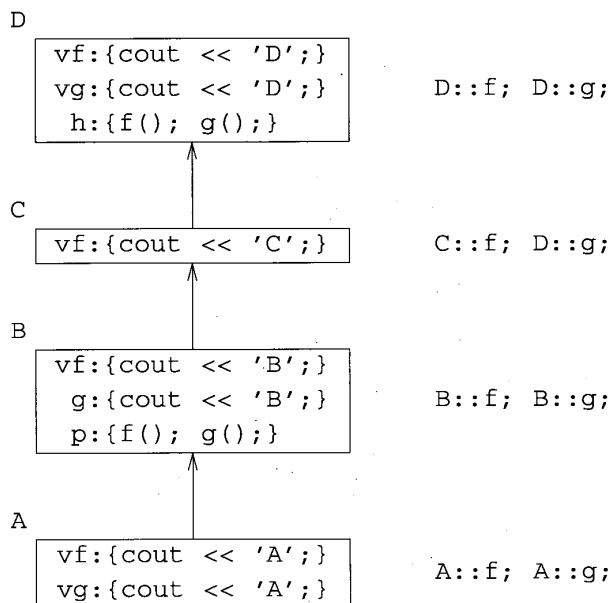


图12-16 虚函数

设a,b,c,和d 分别是A,B,C和D的对象，调用d.h()时需使用D的虚函数表来决定执行哪一个f和g，其输出为 DD。在执行c.h()时，根据C的虚函数表来决定执行哪一个f和g，其输出为CD。b.h()、b.p()、a.h()和a.p()所产生的输出分别是BB、BB、AA和AA。

下面考察多重继承的情况，在这种情况下，A是从两个或两个以上的类中派生而来。例如，考察图12-17a 中的派生有向图，图中A是从B和E中派生而来。现在A有两个虚函数表，第一个对应于派生路径ABCD，第二个对应于路径AEFG。当沿路径ABCD作用于A的对象时使用第一个表；当沿路径AEFG作用于A的对象时使用第二个表。采用与单一继承相同的过程从B的虚函数表中构造第一条路径的虚函数表，同样，也可从E中构造第二条路径的虚函数表。

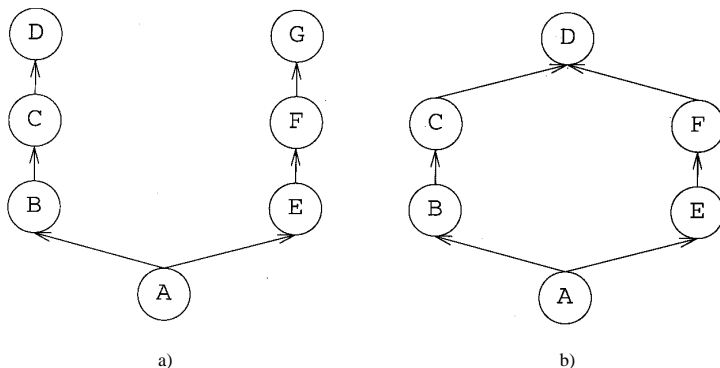


图12-17 派生层次

基于这种构造虚函数表的方法，当从A派生其他的类（比如X）时，需要从A的两个虚函数表中指定一个虚函数表来实现派生。按习惯，指定第一个基类的虚函数表。因此，如果A采用如下语句定义：

```
class A : public B , public E
```

那么无论何时，当构造A的其他派生类的虚函数表时，都将使用路径ABCD的虚函数表作为A的虚函数表。

### 12.9.2 纯虚函数和抽象类

如下所示，如果虚函数被初始化为0：

```
virtual int f (int x,int y )=0 ;
```

则称该虚函数为纯虚函数（pure virtual function）。在它的类说明语句中没有给出实现代码。包含一个纯虚函数的类称为抽象类（abstract class）。如果A是一个抽象类，那么将不会有A类型的对象，因为无法执行A.f()，但是可以有指向A类型对象的指针。

在12.7节中曾提到有向图，无向图，加权有向图和加权无向图都可以被看作一个网络。虽然没有定义相应的网络类，但我们为这四种特殊网络中的每一个网络都定义了两个类，因为必须在描述层对这四种网络进行区分。现在来定义一个抽象类Network（见程序12-15），这个类目前只包含纯虚函数，在后面的小节中将在其中加入一些非虚函数。

程序12-15 抽象类Network

```
class Network {  
public:  
    virtual int Begin(int i) = 0;  
    virtual int NextVertex(int i) = 0;  
    virtual void InitializePos() = 0;  
    virtual void DeactivatePos() = 0;  
};
```

Network 中的函数可施加于所有四种特殊网络。直接使用遍历函数 Begin，NextVertex，InitializePos 和Deactivatepos 可能会失败，因为这些遍历函数是作为虚函数定义的，它们的具体实现取决于对象的类型。

### 12.9.3 虚基类

考察图12-17b 的派生结构，类B、C、D与图12-16中的类相同，将图12-16类A中g 的重定义省略掉就得到图12-17的A。类E和F分别与类B和C不同，区别仅在于E和F只输出字符E和F，而B和C只输出字符B和C。C和F都是从D中派生而来的。如果a 是A的对象，那么a.h（）和a.p（）含义不明。对于第1种情况，不知道应该使用A的两个虚函数表中的哪一个（ABCD和AEFD）；对于第2种情况，不知道是调用a.B::p()还是调用a.E::p()。a.B::h()调用D::h()并使用A的ABCD路径虚函数表，输出结果为AB；a.E::h()产生输出AE。因此，调用的路径决定了使用D的哪一个虚拟函数。

在许多应用中，当对A的对象进行操作时，我们希望不管调用路径是什么，最终执行的是同样的虚函数。通过使D成为C和F的共同虚基类可以做到这一点。不过，在图12-17的例子中，简单地使D成为C和F的虚基类还不够，因为D的虚函数f 和g 在路径ABCD和AEFG上



都被重新定义，因此不知道使用哪一个定义。要消除这种歧义，基类 D 的每一个虚函数最多只能在一条路径上重新定义。例如，假定在类 B 中重新定义 f 来输出 B 且在 A、B 或 C 中没有重新定义 g，在类 E 中重新定义了 g 来输出 E 且在 A、E 或 F 中没有重新定义 f，则在路径 ABCD 和 AEFD 的两个虚函数表中 f 和 g 都是相同的，调用 a.B::h()，a.E::h()，a.E::p() 将产生相同的输出 BE。

使 D 成为 C 和 F 的虚基类的另一个结果是 A 的对象仅含 D 的数据成员的一个拷贝。不管 D 是否是一个虚基类，C 和 F 的对象都包含 D 的数据成员。同样，B 的对象包含 C 和 D 的数据成员，而 E 的对象包含 F 和 D 的数据成员。如果 D 不是一个虚基类，A 的对象将包含路过 B 的 B、C 和 D 的数据成员和路过 E 的 E、F、D 的数据成员。因此，在每一个 A 的对象中包含 2 份 D 的数据成员。当 D 是一个虚基类时，每个 A 的对象将只包含 D 中的 1 份数据成员。

为了避免出现基类数据成员的多个副本，应将基类声明为虚基类。为使 Network 成为 AdjacencyWDigraph 和 LinkBase 的一个虚基类，将类的标题修改如下：

```
class AdjacencyWDigraph : virtual public Network
class LinkBase : virtual public Network
```

这些标题说明了 AdjacencyWDigraph 和 LinkBase 是从 Network 中派生出来的并且 Network 是它们的虚基类。由于其他的类都是从 AdjacencyWDigraph 和 LinkBase 中派生出来的，因此这些类的成员也可以访问 Network 的成员。

在图的应用中，必须把 Network 定义为 AdjacencyWDigraph 和 LinkBase 的一个虚基类，因为还需要定义另外一个类 Undirected，它也是从 Network 派生而来的。类 Undirected 中包含专用于无向图和网络的函数，因此，只能从无向图和网络类中访问这些函数。图 12-18 给出了新的派生结构。

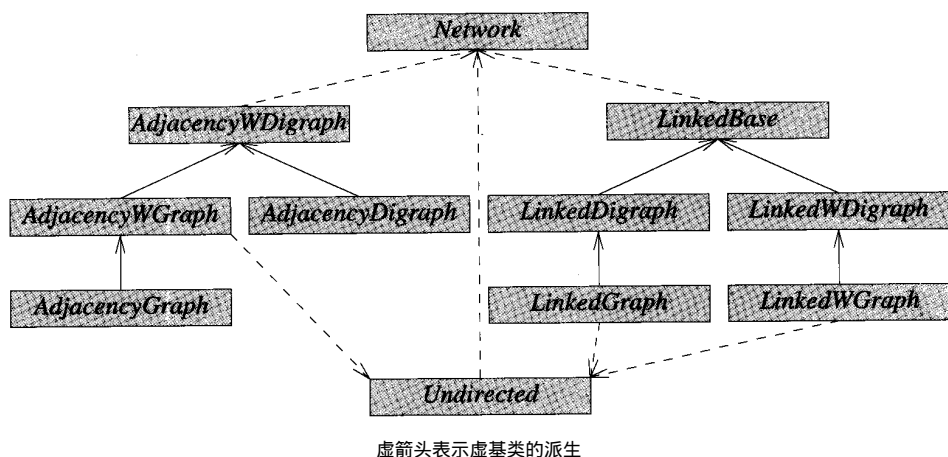


图12-18 包括Undirected的类派生层次

对于链接类，在 LinkBase 中定义了纯虚函数 InitializePos，DeactivatePos，Begin 和 Network 的 NextVertex。假设 G 是 LinkedGraph 类型并且 f 是 Network 的一个成员，假设 LinkBase 和 Undirected 是从 Network 中以非虚拟方式派生出来的，则执行 G.f() 时将出现歧义，因为不知道使用 LinkedGraph 的哪一个虚函数表。如果执行 G.g()，其中 g 是 Undirected 的一个成员，则 g 将调用 f，f 是 Network 的一个成员，因此所使用的路径为 LinkedGraph->Undirected->Network，但 Network 并未定义纯虚遍历函数。通过把 Network 作为 LinkBase 和 Undirected 的

一个虚基类，可以解决这种问题。

#### 12.9.4 抽象类和抽象数据类型

可以用抽象类来说明抽象数据类型。到目前为止，抽象数据类型的描述都是以自然语言的形式给出的。考察抽象数据类型 *LinearList* (ADT2-1)，它是线性表数据结构的非正式描述。该描述给出了线性表必须支持的所有操作，但是无法强制某个具体的线性表实现必须满足这种要求。通过将 *LinearList* 定义成一个抽象类并且要求所有的线性表都从这个抽象类派生而来，就可以使任一个线性表与相应的抽象数据类型保持一致。

对于线性表数据结构，可以使用程序 12-16 的抽象类定义。

程序12-16 抽象类AbstractList

---

```
template<class T>
class AbstractList {
public:
    virtual bool IsEmpty() const = 0;
    virtual int Length() const = 0;
    virtual bool Find(int k, T& x) const = 0; //查找第k个元素，并送入x
    virtual int Search(const T& x) const = 0; //返回x的位置
    virtual AbstractList<T>& Delete(int k, T& x) = 0; //删除第k元素，并将其放入x
    virtual AbstractList<T>& Insert(int k, const T& x) = 0; //紧靠第k个元素之后插入x
    virtual void Output(ostream& out) const = 0;
};
```

---

*AbstractList* 的所有成员函数都是虚函数。由于每一种线性表都必须从 *AbstractList* 派生而来，因此在每种线性表中都必须实现所有的纯虚函数，否则，这种线性表将不可以拥有实例。

为了满足所有线性表都必须从相应抽象类中派生而来的要求，需要将 *LinearList* (见程序 3-1) 和 *Chain* (见程序 3-8) 的类标题改为：

```
class LinearList : AbstractList<T>{
class Chain : AbstractList<T>{
```

此外，还需改变 *Insert* 和 *Delete* 函数的返回类型。对于 *LinearList* 和 *Chain* 类而言，这两个函数的返回类型与 *AbstractList* 有关，因此，必须将语句：

```
LinearList<T>& Delete ( int k, T & x) ;
LinearList<T>& Insert (int k, const T& x) ;
```

和

```
Chain<T>& Delete (int k, T& x) ;
Chain<T>& Insert (int k, const T& x) ;
```

替换为：

```
AbstractList<T>& Delete (int k, T& x) ;
AbstractList<T>& Insert (int k, const T& x) ;
```

并且将语句：

```
LinearList<T> & LinearList<T>::Delete (int k, T& x)
LinearList<T> & LinearList<T>::Insert (int k, const T& x)
```

替换为：

```
AbstractList<T> & LinearList<T>::Delete (int k, T& x)
AbstractList<T> & LinearList<T>::Insert (int k, const T& x)
```

将语句：

```
Chain<T>& Chain<T>::Delete (int k, T& x)
Chain<T>& Chain<T>::Insert (int k, cont T& x)
```

替换为：

```
AbstractList<T> & Chain<T>::Delete(int k, T& x)
AbstractList<T> & Chain<T>::Insert (int k, const T& x)
```

除上述修改以外，其他地方不必做变动。

除了能强制与抽象数据类型描述保持一致以外，使用抽象类还允许编写一些共享函数作为抽象类的成员，而不必为每个派生类分别实现相应的函数。在后面几节中将看到相应的实例。

## 练习

40. 给出一个堆栈 (ADT5-1) 的抽象类定义 `AbstractStack`。修改 `Stack` (见程序 5-2) 和 `LinkedStack` (见程序 5-4)，把它们作为 `AbstractStack` 的派生类。试测试代码的正确性。

41. 给出一个队列 (ADT6-1) 的抽象类定义 `AbstractQueue`。修改 `Queue` (见程序 6-1) 和 `LinkedQueue` (见程序 6-4)，把它们作为 `AbstractQueue` 的派生类。试测试代码的正确性。

## 12.10 图的搜索算法

有关图、有向图和网络函数实在太多，我们无法在这里一一列出。前面已经讨论了其中的一些函数（如寻找路径，寻找生成树，判断无向图是否连通），在后面的章节中还将讨论一些其他的函数。许多函数都要求从一个给定的顶点开始，访问能够到达的所有顶点。（当且仅当存在一条从  $v$  到  $u$  的路径时，顶点  $v$  可到达顶点  $u$ 。）搜索这些顶点的两种标准方法是宽度优先搜索和深度优先搜索。虽然这两种方法都很流行，但比较而言深度优先搜索使用频率更高一些（相应的效率也高一些）。

### 12.10.1 宽度优先搜索

考察图 12-19a 中的有向图。判断从顶点 1 出发可到达的所有顶点的一种方法是首先确定邻接于顶点 1 的顶点集合，这个集合是  $\{2,3,4\}$ 。然后确定邻接于  $\{2,3,4\}$  的新的顶点集合，这个集合是  $\{5,6,7\}$ 。邻接于  $\{5,6,7\}$  的顶点集合为  $\{8,9\}$ ，而不存在邻接于  $\{8,9\}$  的顶点。因此，从顶点 1 出发可到达的顶点集合为  $\{1,2,3,4,5,6,7,8,9\}$ 。

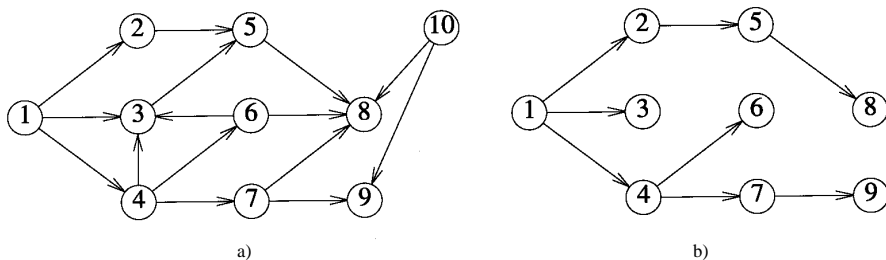


图12-19 宽度优先搜索

这种从一个顶点开始，识别所有可到达顶点的方法叫作宽度优先搜索（ Breadth-First Search, BFS ）。这种搜索可使用队列来实现，图 12-20给出了实现的伪代码。

```
//从顶点v 开始的宽度优先搜索
把顶点v标记为已到达顶点；
初始化队列 Q，其中仅包含一个元素 v；
while (Q 不空) {
    从队列中删除顶点 w；
    令 u 为邻接于 w 的顶点；
    while (u) {
        if ( u 尚未被标记) {
            把 u 加入队列；
            把 u 标记为已到达顶点； }
        u = 邻接于 w 的下一个顶点；
    }
}
```

图12-20 BFS的伪代码

如果将图 12-20的伪代码用于图 12-19a 中， $v=1$ 。因此在第一个 while 循环中，顶点 2,3,4 都将被加入到队列中（假设是按此次序加入的）。在接下来的循环中，2 被从队列中去掉，加入顶点 5；然后删除 3，之后再删除 4，加入 6 和 7；删除 5 并加入 8；删除 6 后不增加；删除 7 后加入 9，最后将 8 和 9 删除，队列成为空队列。过程终止时，顶点 1 到 9 被加上已到达标记。图 12-19b 给出了访问过程中所经历的顶点和边构成的子图。

**定理 12-1** 设  $N$  是一个任意的图、有向图或网络， $v$  是  $N$  中的任意顶点。图 12-20 的伪代码能够标记从  $v$  出发可以到达的所有顶点（包括顶点  $v$ ）。

**证明** 这个定理的证明留作练习 42。

### 12.10.2 类 Network

根据图 12-20 的伪代码，在一个合适的高度，BFS 的执行方式与是否正在处理一个图、有向图、加权图或加权有向图无关，也与所使用的描述方法无关。但是，为了实现下面的语句：

$u =$  邻接于  $w$  的下一个顶点；

必须知道当前正在使用的图类。通过把 BFS 函数作为 Network 类（见程序 12-15）的一个成员函数以及利用图遍历器从一个邻接顶点到达下一个顶点，可以避免为每种图类分别编写不同的代码。

### 12.10.3 BFS 的实现

BFS 的代码（见程序 12-17）与图 12-20 的伪代码非常相似。程序 12-17 假设初始时对于所有顶点有  $reach[i]=0$  并且  $label = 0$ 。算法终止时所有可到达顶点把对应的  $reach[i]$  设置为  $label$ 。

程序 12-17 BFS 代码

```
void Network::BFS(int v, int reach[], int label)
```

```

// 宽度优先搜索
LinkedQueue<int> Q;
InitializePos(); //初始化图遍历器数组
reach[v] = label;
Q.Add(v);
while (!Q.IsEmpty()) {
    int w;
    Q.Delete(w); // 获取一个已标记的顶点
    int u = Begin(w);
    while (u) { // 访问 w的邻接顶点
        if (!reach[u]) { // 一个未曾到达的顶点
            Q.Add(u);
            reach[u] = label; } // 标记已到达该顶点
        u = NextVertex(w); // 下一个与 w邻接的顶点
    }
}
DeactivatePos(); // 释放遍历器数组
}

```

#### 12.10.4 BFS的复杂性分析

从顶点 $v$ 出发, 可达到的每一个顶点都被加上标记, 且每个顶点只加入到队列中一次, 也只从队列中删除一次, 而且它的邻接矩阵中的行或它的邻接链表也只遍历一次。如果有 $s$ 个顶点被标记, 那么当使用邻接矩阵时, 这些操作所需要的时间为 $\Theta(sn)$ , 而使用邻接链表时, 所需时间为 $\Theta(\sum_i d_i^{out})$ 。在后一种情况中, 要对所有被标记的顶点 $i$ 的出度求和。对于无向图/网络来说, 顶点的出度就等于它的度。

现在, 我们想知道, 与为每一种描述都定制一个搜索函数相比, 统一的BFS函数的复杂性是多少。邻接矩阵和邻接链表的搜索函数分别见程序12-18和12-19。

程序12-18 邻接矩阵描述中BFS的直接实现

```

template<class T>
void AdjacencyWDigraph<T>::BFS (int v, int reach[], int label)
// 宽度优先搜索
LinkedQueue<int> Q;
reach[v] = label;
Q.Add(v);
while (!Q.IsEmpty()) {
    int w;
    Q.Delete(w); // 获取一个已标记的顶点
    // 对尚未标记的、邻接自w的顶点进行标记
    for (int u = 1; u <= n; u++)
        if (a[w][u] != NoEdge && !reach[u]) {
            Q.Add(u); // u 未被标记
            reach[u] = label; }
    }
}

```

程序12-19 链接图和有向图中BFS的直接实现

```

void LinkedDigraph::BFS(int v, int reach[], int label)
{// 宽度优先搜索
    LinkedQueue<int> Q;
    reach[v] = label;
    Q.Add(v);
    while (!Q.IsEmpty()) {
        int w;
        Q.Delete(w); // 获取一个已标记的顶点
        // 使用指针p沿着邻接表进行搜索
        ChainNode<int> *p;
        for (p = h[w].First(); p; p = p->link) {
            int u = p->data;
            if (!reach[u]) { // 一个尚未到达的顶点
                Q.Add(u);
                reach[u] = label;
            }
        }
    }
}

```

对于用邻接矩阵描述的含有 50个顶点的无向完全图，Network::BFS的执行时间是AdjacencyWDigraph::BFS 的时间的2.6倍。对于链接描述，统一程序的执行时间是定制程序的4.5倍。

通过删除遍历函数Begin和NextVertex中一些不必要的有效性检查，可以减少统一程序和定制程序之间的差别。在BFS和以后可能定义的Network的其他一些成员中，仅当顶点参数有效时才会调用Begin和NextVertex，因此，可以改进函数Begin和NextVertex，使它们不用执行有效性检查。这样，执行因子4.5将变成3.6（对于50个顶点的完全图）。

如上所述，当使用Network::BFS代替定制程序时，会有一个潜在的巨大代价。但是，Network::BFS也存在不少优点。例如，若使用Network::BFS，则这一份代码即可满足所有的图类描述，但若使用定制程序则必须提供多份不同的代码（每个图类对应一份）。因此，如果要设计新的图类描述并需要实现遍历函数，那么可以不加修改地使用已有的Network成员。

### 12.10.5 深度优先搜索

深度优先搜索（Depth-First Search, DFS）是另一种搜索方法。从顶点 $v$ 出发，DFS按如下过程进行：首先将 $v$ 标记为已到达顶点，然后选择一个与 $v$ 邻接的尚未到达的顶点 $u$ ，如果这样的 $u$ 不存在，搜索中止。假设这样的 $u$ 存在，那么从 $u$ 又开始一个新的DFS。当从 $u$ 开始的搜索结束时，再选择另外一个与 $v$ 邻接的尚未到达的顶点，如果这样的顶点不存在，那么搜索终止。而如果存在这样的顶点，又从这个顶点开始DFS，如此循环下去。

程序12-20给出了Network类的共享成员DFS和私有成员dfs。在DFS的实现过程中，让 $u$ 遍历 $v$ 的所有邻接顶点将更容易。

程序12-20 DFS代码

```

void Network::DFS(int v, int reach[], int label)
{// 深度优先搜索

```

```

InitializePos(); // 初始化图遍历器数组
dfs(v, reach, label); // 执行dfs
DeactivatePos(); // 释放图遍历器数组
}

void Network::dfs(int v, int reach[], int label)
{ // 实际执行深度优先搜索的代码
    reach[v] = label;
    int u = Begin(v);
    while (u) { // u邻接至 v
        if (!reach[u]) dfs(u, reach, label);
        u = NextVertex(v);
    }
}

```

用图12-19a 的有向图来测试DFS。如果 $v=1$ ，那么顶点2,3和4成为 $u$ 的候选。假设赋给 $u$ 的第一个值是2，到达2的边是(1,2)，那么从顶点2 开始一次DFS，将顶点2标记为已到达顶点。这时 $u$  的候选只有顶点5，到达5的边是(2,5)。下面又从5开始进行DFS，将顶点5标记为已到达顶点，根据边(5,8) 可知顶点8也是可到达顶点，将顶点8加上标记。从8开始没有可到达的邻接顶点，因此又返回到顶点5，顶点5也没有新的 $u$ ，因此返回到顶点2，再返回到顶点1。

这时还有两个候选顶点：3和4。假设选中4，边(1,4)存在，从顶点 4开始DFS，将顶点 4标记为已到达顶点。现在顶点 3,6和7成为候选的 $u$ ，假设选中6，当 $u=6$ 时，顶点3是唯一的候选，到达3的边是(6,3)，从3开始DFS，并将3标记为已到达顶点。由于没有与3邻接的新顶点，因此返回到顶点4，从4开始一个 $u=7$ 的DFS，然后到达顶点9，没有与9邻接的其他顶点，这时回到1，没有与1邻接的其他顶点，算法终止。

对于DFS，可以给出一个类似于定理12-1的定理；DFS能够标记出顶点 $v$  和所有可从 $v$  点到达的顶点。

**定理12-2** 设 $N$ 是一个任意的图、有向图或网， $v$  是 $N$ 中任意顶点。对于所有可从顶点 $v$  到达的顶点（包括 $v$ ），调用DFS（ $v, reach, label$ ）后，可得 $reached[i]=label$ 。

**证明** 这个定理的证明留作练习43。

可以验证DFS与BFS有相同的时间和空间复杂性。不过，使DFS占用最大空间(递归栈空间)的图却是使BFS占用最小空间(队列空间)的图，而使BFS占用最大空间的图则是使DFS占用最小空间的图。图12-21中给出了能使DFS和BFS产生最好和最坏性能的图例。

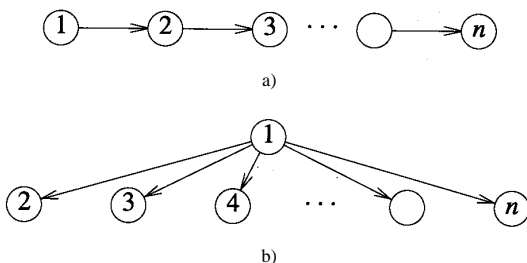


图12-21 产生最好和最坏空间复杂性的图例

- a) DepthFirstSearch(1) 的最坏情况；BreadthFirst Search(1) 的最好情况  
 b) DepthFirstSearch(1) 的最好情况；BreadthFirst Search(1) 的最坏情况



## 练习

42. 证明定理 12-1。
43. 证明定理 12-2。
44. 编写 PackedAdjGraph 类的遍历函数。

## 12.11 应用

## 12.11.1 寻找路径

若从顶点  $v$  开始搜索（宽度或深度优先）且到达顶点  $w$  时终止搜索，则可以找到一条从顶点  $v$  到达顶点  $w$  的路径（例 12-1）。要实际构造这条路径，需要记住从一个顶点到下一个顶点的边。对于路径问题，所需要的边的集合已隐含在深度优先的递归过程中，因此可以很容易地利用深度优先策略设计一个寻找路径程序。完成顶点  $w$  的标记之后展开递归，可以反向建立起从  $w$  到  $v$  的路径。FindPath 的代码如程序 12-21 所示。这个程序要求把 Vertices() 定义为 Network 的一个虚拟成员。

FindPath 的输入参数是路径的开始顶点 ( $v$ ) 和目标顶点 ( $w$ )。如果没有从  $v$  到  $w$  的路径，FindPath 返回 false；否则返回 true。当找到路径时，用参数 length 返回路径长度（路径中边的条数），用顶点数组  $p[0 : \text{length}]$  返回路径，其中  $p[0]=v$  且  $p[\text{length}]=w$ 。

FindPath 首先检验  $v=w$  情况。在这种情况下，返回一个长度为 0 的路径。如果  $v \neq w$ ，则调用图的遍历函数 InitializePos，然后 FindPath 产生并初始化一个数组 reach，路径的 DFS 实际上是由 Network 的私有成员 findpath 完成的，当且仅当没有路径时，findpath 返回 false。函数 findpath 是一个修改过的 DFS，它对标准的 DFS 作了如下两点修改：1) 一旦到达了目标顶点  $w$ ，findpath 将不再继续搜索可到达顶点；2) findpath 将开始顶点  $v$  到当前顶点  $u$  路径中的顶点记录到数组 path 中。

Findpath 与 DFS 具有相同的复杂性。

程序 12-21 在图中寻找一个路径

```
bool Network::FindPath (int v, int w, int &length, int path[])
// 寻找一条从 v 到 w 的路径, 返回路径的长度, 并将路径存入数组 path[0:length]
// 如果不存在路径, 则返回 false

// 路径中的第一个顶点总是 v
path[0] = v;
length = 0; // 当前路径的长度
if (v == w) return true;

// 为路径的递归搜索进行初始化
int n = Vertices();
InitializePos(); // 遍历器
int *reach = new int [n+1];
for (int i = 1; i <= n; i++)
    reach[i] = 0;

// 搜索路径
```

```

bool x = findPath(v, w, length, path, reach);

DeactivatePos();
delete [] reach;
return x;
}

bool Network::findPath(int v, int w, int &length, int path[], int reach[])
{// 实际搜索v到w的路径, 其中 v != w.
// 按深度优先方式搜索一条到达w的路径
reach[v] = 1;
int u = Begin(v);
while (u) {
    if (!reach[u]) {
        length++;
        path[length] = u; // 将u 加入path
        if (u == w) return true;
        if (findPath(u, w, length, path, reach))
            return true;
        // 不存在从 u 到 w 的路径
        length--; //删除u
    }
    u = NextVertex(v);}
return false;
}

```

### 12.11.2 连通图及其构件

通过从任意顶点开始执行DFS或BFS, 并且检验所有顶点是否被标记为已到达顶点, 可以判断一个无向图G是否连通。虽然这个算法只是直接检验BFS中从开始顶点到其他每一个顶点之间是否存在一条路径, 但对于判断两个顶点之间是否存在一条路径来说, 这种检验已经足够了。假设*i*是搜索的开始顶点并且搜索到达了图中的所有顶点, 利用*i*到*u*的反向路径及*i*到*v*的路径, 可以构造任意两个顶点*u*和*v*之间的路径。如果图不连通, 则函数Connected (见程序12-22) 返回false, 否则返回true。由于连通的概念仅针对无向图和网络而言, 因此, 可以定义一个新类Undirected, 函数Connected是Undirected的一个成员。图12-18给出了Undirected的类定义。

程序12-22 确定无向图是否连通

```

class Undirected : virtual public Network {
public:
    bool Connected();
};

bool Undirected::Connected()
{// 当且仅当图是连通的, 则返回 true

    int n = Vertices();

```

```
// 置所有顶点为未到达顶点
int *reach = new int [n+1];
for (int i = 1; i <= n; i++)
    reach[i] = 0;

// 对从顶点1出发可到达的顶点进行标记
DFS(1, reach, 1);

// 检查是否所有顶点都已经被标记
for (int i = 1; i <= n; i++)
    if (!reach[i]) return false;
return true;
}
```

从顶点 $i$ 可到达的顶点的集合 $C$ 与连接 $C$ 中顶点的边称为连通构件 (connected component)。图12-1b 的图中有2个连通构件，一个由顶点 $\{1,2,3\}$ 和边 $\{(1,2), (1,3)\}$ 组成，另一个由其他顶点和边组成。在构件标识问题 (component-labeling problem) 中，对图中的顶点进行标识，当且仅当2个顶点属于同一构件时，分配给它们相同的标号。在图 12-1b 的例子中，顶点1和2标识为标号1，而剩下的顶点标识为标号2。

可以通过反复调用DFS或BFS算法来标识构件。从每一个尚未标识的顶点开始进行搜索，并用新的标号标识新到达的顶点。函数LabelComponents (见程序12-23) 解决了构件标识问题。该函数返回图中构件的数目，并将构件标号返回至数组  $L$  中。在程序12-23中，如果用DFS来取代BFS，也能得到相同结果。当用邻接矩阵来描述图时，程序 12-23的复杂性是  $\Theta(n^2)$ ；而用邻接链表时，复杂性为  $\Theta(n+e)$ 。

程序12-23 构件标识

```
int Undirected::LabelComponents(int L[])
{
    // 构件标识
    // 返回构件的数目，并用 L[1:n]表示构件标号

    int n = Vertices();

    // 初始时，所有顶点都不属于任何构件
    for (int i = 1; i <= n; i++)
        L[i] = 0;

    int label = 0; // 最后一个构件的ID
    // 识别构件
    for (int i = 1; i <= n; i++)
        if (!L[i]) { // 未到达的顶点
            // 顶点 i 属于一个新的构件
            label++;
            BFS(i, L, label); // 标记新构件
        }

    return label;
}
```

## 12.11.3 生成树

在一个  $n$  顶点的连通无向图中，如果从任一顶点开始进行 BFS，那么从定理 12-1 可知，所有顶点都将被加上标记，并且在 `Network::BFS` (见程序 12-17) 的内层 `while` 循环中正好有  $n-1$  个顶点是可到达的。在该循环中，若到达一个新顶点  $u$ ，则相应的边为  $(w,u)$ ，这样边的数目正

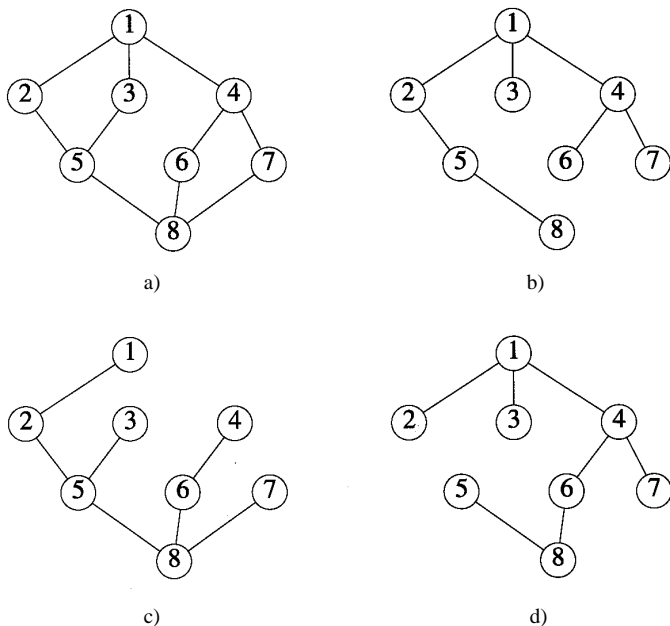


图12-22 图及其宽度优先生成树

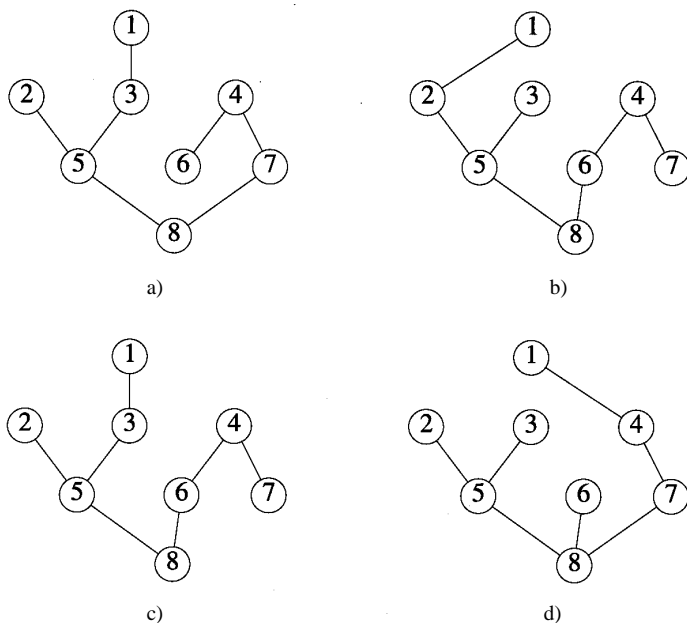


图12-23 图12-22a 的一些深度优先生成树

好是 $n-1$ 。由于所得到的边的集合中包含一条从 $v$ 到图中其他每个顶点的路径，因此它构成了一个连通子图，该子图即为 $G$ 的生成树。

考察图12-22a中的图，如果从顶点1开始进行BFS，那么用来到达以前未到达顶点的边是 $\{(1,2), (1,3), (1,4), (2,5), (4,6), (4,7), (5,8)\}$ ，这个边集合即对应于图12-22b中的生成树。

宽度优先生成树（breadth-first spanning tree）是按BFS所得到的生成树。可以验证图12-22b、c和d中的生成树都是图12-22a的宽度优先生成树。（图12-22c和d分别是分别从顶点8和6开始搜索而得到的。）

当在一个无向连通图或网络中执行BFS时，到达新顶点的边正好有 $n-1$ 条，这些边组成的子图也是一个生成树，用这种方法所得到的生成树叫作深度优先生成树（depth-first spanning tree）。图12-23给出了图12-22a的一些深度优先生成树。

## 练习

45. 根据图12-1a，完成以下练习：

- 1) 从顶点1开始产生一个宽度优先生成树。
- 2) 从顶点3开始产生一个宽度优先生成树。
- 3) 从顶点1开始产生一个深度优先生成树。
- 4) 从顶点3开始产生一个深度优先生成树。

46. 编写共享成员 `Undirected::BSpanningTree(i,BT)`，该函数从一个连通无向图或网络中的顶点 $i$ 开始寻找一个宽度优先生成树。若因内存问题导致搜索失败，程序应能引发异常；若因没有生成树（图是非连通的）而导致失败，则返回 `false`；否则返回 `true`。找到生成树时，将边返回到数组 `BT` 中。定义 `BT` 的数据类型。

47. 针对 `Undirected::DSpanningTree(i,BT)` 完成练习46，该函数从顶点 $i$ 开始寻找一个深度优先生成树。

48. 编写共享成员 `Network::Cycle()`，用于确定网络中是否存在一个（有向）环路。可基于DFS或BFS来实现。

- 1) 证明代码的正确性。
- 2) 指出程序的时间和空间复杂性。

49. 设 $G$ 是一个无向连通图或网络。编写函数 `Undirected::Bipartite(L)`，如果 $G$ 不是一个二分图（见例子12-3），则函数返回 `false`，否则返回 `true`。当 $G$ 是二分图时，函数还得在 $L$ 中返回一个标号，如对于一个子集中的顶点，有  $L[i]=1$ ，而对于另一个子集中的顶点， $L[i]=2$ 。如果 $G$ 有 $n$ 个顶点且用矩阵描述，那么程序的复杂性应为  $\Theta(n^2)$ 。而如果用链表来描述 $G$ ，则复杂性应为  $\Theta(n+e)$ 。（提示：执行多次BFS，每次均从目前未到达的顶点开始，将这个顶点分配到集合1；与该顶点邻接的顶点分配至集合2；与集合2中顶点邻接的顶点再放入集合1，如此进行下去。其间需检查分配冲突。）

50.  $G$ 是一个无向图或网络，它的传递闭包（transitive closure）是一个0/1数组 `TC`，当且仅当 $G$ 中存在一条边数大于1的从 $i$ 到 $j$ 的路径时， $TC[i][j]=1$ 。编写一个函数 `Undirected::TransitiveClosure(TC)`，计算 $G$ 的传递闭包矩阵。函数的复杂性应为  $\Theta(n^2)$ ，其中 $n$ 是 $G$ 的顶点数目。（提示：采用构件标识策略。）

51. 若 $G$ 是有向图，针对 `Network::TransitiveClosure(TC)` 完成练习50。函数的复杂性是多少？

## 第三部分 算法设计方法

### 第13章 贪婪算法

离开了数据结构的世界，现在进入算法设计方法的世界。

从本章开始，我们来研究一些算法设计方法。虽然设计一个好的求解算法更像是一门艺术，而不像是技术，但仍然存在一些行之有效的能够用于解决许多问题的算法设计方法，你可以使用这些方法来设计算法，并观察这些算法是如何工作的。一般情况下，为了获得较好的性能，必须对算法进行细致的调整。但是在某些情况下，算法经过调整之后性能仍无法达到要求，这时就必须寻求另外的方法来求解该问题。

本书的第13~17章提供了五种基本的算法设计方法：贪婪算法、分而治之算法、动态规划、回溯和分枝定界。而其他的常用高级方法如：线性规划、整数规划、遗传算法、模拟退火等则没有提及。有关这些方法的详细描述请参见相关书籍。

本章首先引入最优化的概念，然后介绍一种直观的问题求解方法：贪婪算法。最后，应用该算法给出货箱装船问题、背包问题、拓扑排序问题、二分覆盖问题、最短路径问题、最小代价生成树等问题的求解方案。

#### 13.1 最优化问题

本章及后续章节中的许多例子都是最优化问题（optimization problem），每个最优化问题都包含一组限制条件（constraint）和一个优化函数（optimization function），符合限制条件的问题求解方案称为可行解（feasible solution），使优化函数取得最佳值的可行解称为最优解（optimal solution）。

例13-1 [渴婴问题] 有一个非常渴的、聪明的小婴儿，她可能得到的东西包括一杯水、一桶牛奶、多罐不同种类的果汁、许多不同的装在瓶子或罐子中的苏打水，即婴儿可得到  $n$  种不同的饮料。根据以前关于这  $n$  种饮料的不同体验，此婴儿知道这其中某些饮料更合自己的胃口，因此，婴儿采取如下方法为每一种饮料赋予一个满意度值：饮用 1 盎司第  $i$  种饮料，对它作出相对评价，将一个数值  $s_i$  作为满意度赋予第  $i$  种饮料。

通常，这个婴儿都会尽量饮用具有最大满意度值的饮料来最大限度地满足她解渴的需要，但是不幸的是：具有最大满意度值的饮料有时并没有足够的量来满足此婴儿解渴的需要。设  $a_i$  是第  $i$  种饮料的总量（以盎司为单位），而此婴儿需要  $t$  盎司的饮料来解渴，那么，需要饮用  $n$  种不同的饮料各多少量才能满足婴儿解渴的需求呢？

设各种饮料的满意度已知。令  $x_i$  为婴儿将要饮用的第  $i$  种饮料的量，则需要解决的问题是：找到一组实数  $x_i$  ( $1 \leq i \leq n$ )，使  $\sum_{i=1}^n s_i x_i$  最大，并满足： $\sum_{i=1}^n x_i = t$  及  $0 \leq x_i \leq a_i$ 。

需要指出的是：如果  $\sum_{i=1}^n a_i < t$ ，则不可能找到问题的求解方案，因为即使喝光所有的饮料也不能使婴儿解渴。

对上述问题精确的数学描述明确地指出了程序必须完成的工作, 根据这些数学公式, 可以对输入 / 输出作如下形式的描述:

输入:  $n, t, s_i, a_i$  (其中  $1 \leq i \leq n$ ,  $n$  为整数,  $t, s_i, a_i$  为正实数)。

输出: 实数  $x_i$  ( $1 \leq i \leq n$ ), 使  $\sum_{i=1}^n s_i x_i$  最大且  $\sum_{i=1}^n x_i = t$  ( $0 \leq x_i \leq a_i$ )。如果  $\sum_{i=1}^n a_i < t$ , 则输出适当的错误信息。

在这个问题中, 限制条件是  $\sum_{i=1}^n x_i = t$  且  $0 \leq x_i \leq a_i, 1 \leq i \leq n$ 。而优化函数是  $\sum_{i=1}^n s_i x_i$ 。任何满足限制条件的一组实数  $x_i$  都是可行解, 而使  $\sum_{i=1}^n s_i x_i$  最大的可行解是最优解。

例13-2 [装载问题] 有一艘大船准备用来装载货物。所有待装货物都装在货箱中且所有货箱的大小都一样, 但货箱的重量都各不相同。设第  $i$  个货箱的重量为  $w_i$  ( $1 \leq i \leq n$ ), 而货船的最大载重量为  $c$ , 我们的目的是在货船上装入最多的货物。

这个问题可以作为最优化问题进行描述: 设存在一组变量  $x_i$ , 其可能取值为0或1。如  $x_i$  为0, 则货箱  $i$  将不被装上船; 如  $x_i$  为1, 则货箱  $i$  将被装上船。我们的目的是找到一组  $x_i$ , 使它满足限制条件  $\sum_{i=1}^n w_i x_i \leq c$  且  $x_i \in \{0, 1\}, 1 \leq i \leq n$ 。相应的优化函数是  $\sum_{i=1}^n x_i$ 。

满足限制条件的每一组  $x_i$  都是一个可行解, 能使  $\sum_{i=1}^n x_i$  取得最大值的方案是最优解。

例13-3 [最小代价通讯网络] 这个问题曾在例12-2介绍过。城市及城市之间所有可能的通信连接可被视作一个无向图, 图的每条边都被赋予一个权值, 权值表示建成由这条边所表示的通信连接所要付出的代价。包含图中所有顶点 (城市) 的连通子图都是一个可行解。设所有的权值都非负, 则所有可能的可行解都可表示成无向图的一组生成树, 而最优解是其中具有最小代价的生成树。

在这个问题中, 需要选择一个无向图中的边集合的子集, 这个子集必须满足如下限制条件: 所有的边构成一个生成树。而优化函数是子集中所有边的权值之和。

## 13.2 算法思想

在贪婪算法 (greedy method) 中采用逐步构造最优解的方法。在每个阶段, 都作出一个看上去最优的决策 (在一定的标准下)。决策一旦作出, 就不可再更改。作出贪婪决策的依据称为贪婪准则 (greedy criterion)。

例13-4 [找零钱] 一个小孩买了价值少于1美元的糖, 并将1美元的钱交给售货员。售货员希望用数目最少的硬币找给小孩。假设提供了数目不限的面值为25美分、10美分、5美分、及1美分的硬币。售货员分步骤组成要找的零钱数, 每次加入一个硬币。选择硬币时所采用的贪婪准则如下: 每一次选择应使零钱数尽量增大。为保证解法的可行性 (即: 所给的零钱等于要找的零钱数), 所选择的硬币不应使零钱总数超过最终所需的数目。

假设需要找给小孩67美分, 首先入选的是两枚25美分的硬币, 第三枚入选的不能是25美分的硬币, 否则硬币的选择将不可行 (零钱总数超过67美分), 第三枚应选择10美分的硬币, 然后是5美分的, 最后加入两个1美分的硬币。

贪婪算法有种直觉的倾向, 在找零钱时, 直觉告诉我们应使找出的硬币数目最少 (至少是接近最少的数目)。可以证明采用上述贪婪算法找零钱时所用的硬币数目的确最少 (见练习1)。

例13-5 [机器调度] 现有  $n$  件任务和无限多台的机器, 任务可以在机器上得到处理。每件任务的开始时间为  $s_i$ , 完成时间为  $f_i$ ,  $s_i < f_i$ 。  $[s_i, f_i]$  为处理任务  $i$  的时间范围。两个任务  $i, j$  重叠是



指两个任务的时间范围区间有重叠，而并非是指  $i, j$  的起点或终点重合。例如：区间  $[1, 4]$  与区间  $[2, 4]$  重叠，而与区间  $[4, 7]$  不重叠。一个可行的任务分配是指在分配中没有两件重叠的任务分配给同一台机器。因此，在可行的分配中每台机器在任何时刻最多只处理一个任务。最优分配是指使用的机器最少的可行分配方案。

假设有  $n=7$  件任务，标号为  $a$  到  $g$ 。它们的开始与完成时间如图 13-1a 所示。若将任务  $a$  分给机器  $M1$ ，任务  $b$  分给机器  $M2$ ，...，任务  $g$  分给机器  $M7$ ，这种分配是可行的分配，共使用了七台机器。但它不是最优分配，因为有其他分配方案可使利用的机器数目更少，例如：可以将任务  $a$ 、 $b$ 、 $d$  分配给同一台机器，则机器的数目降为五台。

一种获得最优分配的贪婪方法是逐步分配任务。每步分配一件任务，且按任务开始时间的非递减次序进行分配。若已经至少有一件任务分配给某台机器，则称这台机器是旧的；若机器非旧，则它是新的。在选择机器时，采用以下贪婪准则：根据欲分配任务的开始时间，若此时有旧的机器可用，则将任务分给旧的机器。否则，将任务分配给一台新的机器。

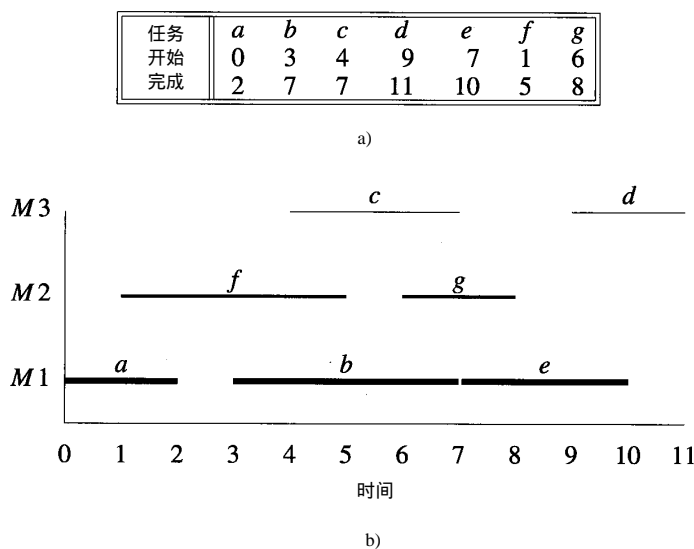


图13-1 任务及三台机器的调度

a) 7个任务 b) 调度

根据例子中的数据，贪婪算法共分为  $n=7$  步，任务分配的顺序为  $a$ 、 $f$ 、 $b$ 、 $c$ 、 $g$ 、 $e$ 、 $d$ 。第一步没有旧机器，因此将  $a$  分配给一台新机器（比如  $M1$ ）。这台机器在 0 到 2 时刻处于忙状态（如图 13-1b 所示）。在第二步，考虑任务  $f$ 。由于当  $f$  启动时旧机器仍处于忙状态，因此将  $f$  分配给一台新机器（设为  $M2$ ）。第三步考虑任务  $b$ ，由于旧机器  $M1$  在  $S_b=3$  时刻已处于闲状态，因此将  $b$  分配给  $M1$  执行， $M1$  下一次可用时刻变成  $f_b=7$ ， $M2$  的可用时刻变成  $f_f=5$ 。第四步，考虑任务  $c$ 。由于没有旧机器在  $S_c=4$  时刻可用，因此将  $c$  分配给一台新机器（ $M3$ ），这台机器下一次可用时间为  $f_c=7$ 。第五步考虑任务  $g$ ，将其分配给机器  $M2$ ，第六步将任务  $e$  分配给机器  $M1$ ，最后在这第七步，任务  $d$  分配给机器  $M3$ 。（注意：任务  $d$  也可分配给机器  $M2$ ）。

上述贪婪算法能导致最优机器分配的证明留作练习（练习 7）。可按如下方式实现一个复杂性为  $O(n \log n)$  的贪婪算法：首先采用一个复杂性为  $O(n \log n)$  的排序算法（如堆排序）按  $S_i$  的递增次序排列各个任务，然后使用一个关于旧机器可用时间的最小堆。

例13-6 [最短路径] 给出一个如图13-2所示的有向网络，路径的长度定义为路径所经过的各边的耗费之和。要求找一条从初始顶点 $s$ 到达目的顶点 $d$ 的最短路径。

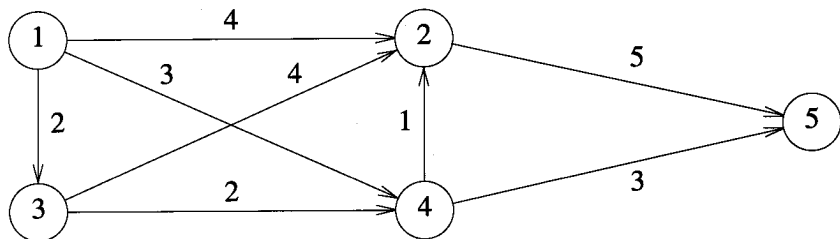


图13-2 有向图

贪婪算法分步构造这条路径，每一步在路径中加入一个顶点。假设当前路径已到达顶点 $q$ ，且顶点 $q$ 并不是目的顶点 $d$ 。加入下一个顶点所采用的贪婪准则为：选择离 $q$ 最近且目前不在路径中的顶点。

这种贪婪算法并不一定能获得最短路径。例如，假设在图13-2中希望构造从顶点1到顶点5的最短路径，利用上述贪婪算法，从顶点1开始并寻找目前不在路径中的离顶点1最近的顶点。到达顶点3，长度仅为2个单位，从顶点3可以到达的最近顶点为4，从顶点4到达顶点2，最后到达目的顶点5。所建立的路径为1,3,4,2,5，其长度为10。这条路径并不是有向图中从1到5的最短路径。事实上，有几条更短的路径存在，例如路径1,4,5的长度为6。

根据上面三个例子，回想一下前几章所考察的一些应用，其中有几种算法也是贪婪算法。例如，9.5.3节中的霍夫曼树算法，利用 $n-1$ 步来建立最小加权外部路径的二叉树，每一步都将两棵二叉树合并为一棵，算法中所使用的贪婪准则为：从可用的二叉树中选出权重最小的两棵。

9.5.2节的LPT调度规则也是一种贪婪算法，它用 $n$ 步来调度 $n$ 个作业。首先将作业按时间长短排序，然后在每一步中为一个任务分配一台机器。选择机器所利用的贪婪准则为：使目前的调度时间最短。将新作业调度到最先完成的机器上（即最先空闲的机器）。

注意到在9.5.2节中的机器调度问题中，贪婪算法并不能保证最优，然而，那是一种直觉的倾向且一般情况下结果总是非常接近最优值。它利用的规则就是在实际环境中希望人工机器调度所采用的规则。算法并不保证得到最优结果，但通常所得结果与最优解相差无几，这种算法也称为启发式方法（heuristics）。因此LPT方法是一种启发式机器调度方法。定理9-2陈述了LPT调度的完成时间与最佳调度的完成时间之间的关系，因此LPT启发式方法具有限定性能（bounded performance）。具有限定性能的启发式方法称为近似算法（approximation algorithm）。

10.5.1节阐述了解决箱子装载问题的几种具有限定性能的启发式方法（即近似算法），每一种启发式方法都是贪婪启发法，9.5.2节的LPT法也是一种贪婪启发法。所有这些启发式方法都具有直觉倾向，并且在实际应用中，这些方法所得到的结果比使用9.5.2节中限定方法所得到的结果更接近最优解。

本章的其余部分将介绍几种贪婪算法的应用。在有些应用中，贪婪算法所产生的结果总是最优的解决方案。但对其他一些应用，生成的算法只是一种启发式方法，可能是也可能不是近似算法。

## 练习

1. 证明找零钱问题（例13-4）的贪婪算法总能产生具有最少硬币数的零钱。
2. 考虑例13-4的找零钱问题，假设售货员只有有限的 25美分，10美分，5美分和1美分的硬币，给出一种找零钱的贪婪算法。这种方法总能找出具有最少硬币数的零钱吗？证明结论。
3. 扩充例13-4的算法，假定售货员除硬币外还有50, 20, 10, 5, 和1美元的纸币，顾客买价格为 $x$  美元和 $y$  美分的商品时所付的款为 $u$  美元和 $v$  美分。算法总能找出具有最少纸币与硬币数目的零钱吗？证明结论。
4. 编写一个C++程序实现例13-4的找零钱算法。假设售货员具有面值为100, 20, 10, 5和1美元的纸币和各种硬币。程序可包括输入模块（即输入所买商品的价格及顾客所付的钱数），输出模块（输出零钱的数目及要找的各种货币的数目）和计算模块（计算怎样给出零钱）。
5. 假设某个国家所使用硬币的币值为14, 2, 5和1分，则例13-4的贪婪算法总能产生具有最少硬币数的零钱吗？证明结论。
6. 1) 证明例13-5的贪婪算法总能找到最优任务分配方案。  
2) 实现这种算法，使其复杂性为 $O(n \log n)$ （提示：根据完成时间建立最小堆）。
- \*7. 考察例13-5的机器调度问题。假定仅有一台机器可用，那么将选择最大数量的任务在这台机器上执行。例如，所选择的最大任务集合为 $\{a, b, e\}$ 。解决这种任务选择问题的贪婪算法可按步骤选择任务，每步选择一个任务，其贪婪准则如下：从剩下的任务中选择具有最小的完成时间且不会与现有任务重叠的任务。
  - 1) 证明上述贪婪算法能够获得最优选择。
  - 2) 实现该算法，其复杂性应为 $O(n \log n)$ 。（提示：采用一个完成时间的最小堆）

## 13.3 应用

## 13.3.1 货箱装船

这个问题来自例13-2。船可以分步装载，每步装一个货箱，且需要考虑装载哪一个货箱。根据这种思想可利用如下贪婪准则：从剩下的货箱中，选择重量最小的货箱。这种选择次序可以保证所选的货箱总重量最小，从而可以装载更多的货箱。根据这种贪婪策略，首先选择最轻的货箱，然后选次轻的货箱，如此下去直到所有货箱均装上船或船上不能再容纳其他任何一个货箱。

例13-7 假设 $n=8$ ,  $[w_1, \dots, w_8]=[100, 200, 50, 90, 150, 50, 20, 80]$ ,  $c=400$ 。利用贪婪算法时，所考察货箱的顺序为7, 3, 6, 8, 4, 1, 5, 2。货箱7, 3, 6, 8, 4, 1的总重量为390个单位且已被装载，剩下的装载能力为10个单位，小于剩下的任何一个货箱。在这种贪婪解算法中得到 $[x_1, \dots, x_8]=[1, 0, 1, 1, 0, 1, 1, 1]$ 且 $x_i=6$ 。

定理13-1 利用贪婪算法能产生最佳装载。

证明 可以采用如下方式来证明贪婪算法的最优性：令 $x=[x_1, \dots, x_n]$ 为用贪婪算法获得的解，令 $y=[y_1, \dots, y_n]$ 为任意一个可行解，只需证明 $\sum_{i=1}^n x_i \leq \sum_{i=1}^n y_i$ 。不失一般性，可以假设货箱都排好了序：即 $w_i \leq w_{i+1}$  ( $1 \leq i < n$ )。然后分几步将 $y$ 转化为 $x$ ，转换过程中每一步都产生一个可行的新 $y$ ，且 $\sum_{i=1}^n y_i$ 大于等于未转化前的值，最后便可证明 $\sum_{i=1}^n x_i \leq \sum_{i=1}^n y_i$ 。

根据贪婪算法的工作过程，可知在 $[0, n]$ 的范围内有一个 $k$ ，使得 $x_i=1$ ,  $i \leq k$ 且 $x_i=0$ ,  $i > k$ 。寻

找 $[1, n]$ 范围内最小的整数 $j$ , 使得 $x_j = y_j$ 。若没有这样的 $j$ 存在, 则 $\sum_{i=1}^n x_i = \sum_{i=1}^n y_i$ 。如果有这样的 $j$ 存在, 则 $j < k$ , 否则 $y$ 就不是一个可行解, 因为 $x_j = y_j$ ,  $x_j = 1$ 且 $y_j = 0$ 。令 $y_j = 1$ , 若结果得到的 $y$ 不是可行解, 则在 $[j+1, n]$ 范围内必有一个 $l$ 使得 $y_l = 1$ 。令 $y_l = 0$ , 由于 $w_j = w_l$ , 则得到的 $y$ 是可行的。而且, 得到的新 $y$ 至少与原来的 $y$ 具有相同数目的1。

经过数次这种转化, 可将 $y$ 转化为 $x$ 。由于每次转化产生的新 $y$ 至少与前一个 $y$ 具有相同数目的1, 因此 $x$ 至少与初始的 $y$ 具有相同的数目1。

货箱装载算法的C++代码实现见程序13-1。由于贪婪算法按货箱重量递增的顺序装载, 程序13-1首先利用间接寻址排序函数IndirectSort对货箱重量进行排序(见3.5节间接寻址的定义), 随后货箱便可按重量递增的顺序装载。由于间接寻址排序所需的时间为 $O(n \log n)$ (也可利用9.5.1节的堆排序及第14章的归并排序), 算法其余部分所需时间为 $O(n)$ , 因此程序13-1的总的复杂性为 $O(n \log n)$ 。

程序13-1 货箱装船

```
template<class T>
void ContainerLoading(int x[], T w[], T c, int n)
{
    // 货箱装船问题的贪婪算法
    // x[i] = 1 当且仅当货箱 i 被装载, 1 ≤ i ≤ n
    // c 是船的容量, w 是货箱的重量

    // 对重量按间接寻址方式排序
    // t 是间接寻址表
    int *t = new int [n+1];
    IndirectSort(w, t, n);
    // 此时, w[t[i]] ≤ w[t[i+1]], 1 ≤ i ≤ n

    // 初始化 x
    for (int i = 1; i ≤ n; i++)
        x[i] = 0;

    // 按重量次序选择物品
    for (i = 1; i ≤ n && w[t[i]] ≤ c; i++) {
        x[t[i]] = 1;
        c -= w[t[i]]; // 剩余容量
    }

    delete [] t;
}
```

### 13.3.2 0/1 背包问题

在0/1背包问题中, 需对容量为 $c$ 的背包进行装载。从 $n$ 个物品中选取装入背包的物品, 每件物品 $i$ 的重量为 $w_i$ , 价值为 $p_i$ 。对于可行的背包装载, 背包中物品的总重量不能超过背包的容量, 最佳装载是指所装入的物品价值最高, 即 $\sum_{i=1}^n p_i x_i$ 取得最大值。约束条件为 $\sum_{i=1}^n w_i x_i \leq c$ 和 $x_i \in [0, 1] (1 \leq i \leq n)$ 。

在这个表达式中, 需求出 $x_i$ 的值。 $x_i = 1$ 表示物品 $i$ 装入背包中,  $x_i = 0$ 表示物品 $i$ 不装入背包。

0/1背包问题是一个一般化的货箱装载问题，即每个货箱所获得的价值不同。货箱装载问题转化为背包问题的形式为：船作为背包，货箱作为可装入背包的物品。

例13-8 在杂货店比赛中你获得了第一名，奖品是一车免费杂货。店中有  $n$  种不同的货物。规则规定从每种货物中最多只能拿一件，车子的容量为  $c$ ，物品  $i$  需占用  $w_i$  的空间，价值为  $p_i$ 。你的目标是使车中装载的物品价值最大。当然，所装货物不能超过车的容量，且同一种物品不得拿走多件。这个问题可仿照0/1背包问题进行建模，其中车对应于背包，货物对应于物品。

0/1背包问题有好几种贪婪策略，每个贪婪策略都采用多步过程来完成背包的装入。在每一步过程中利用贪婪准则选择一个物品装入背包。一种贪婪准则为：从剩余的物品中，选出可以装入背包的价值最大的物品，利用这种规则，价值最大的物品首先被装入（假设有足够容量），然后是下一个价值最大的物品，如此继续下去。这种策略不能保证得到最优解。例如，考虑  $n=2$ ,  $w=[100,10,10]$ ,  $p=[20,15,15]$ ,  $c=105$ 。当利用价值贪婪准则时，获得的解为  $x=[1,0,0]$ ，这种方案的总价值为20。而最优解为  $[0,1,1]$ ，其总价值为30。

另一种方案是重量贪婪准则是：从剩下的物品中选择可装入背包的重量最小的物品。虽然这种规则对于前面的例子能产生最优解，但在一般情况下则不一定能得到最优解。考虑  $n=2$ ,  $w=[10,20]$ ,  $p=[5,100]$ ,  $c=25$ 。当利用重量贪婪策略时，获得的解为  $x=[1,0]$ ，比最优解  $[0,1]$  要差。

还可以利用另一方案，价值密度  $p_i/w_i$  贪婪算法，这种选择准则为：从剩余物品中选择可装入包的  $p_i/w_i$  值最大的物品，这种策略也不能保证得到最优解。利用此策略试解  $n=3$ ,  $w=[20,15,15]$ ,  $p=[40,25,25]$ ,  $c=30$  时的最优解。

我们不必因所考察的几个贪婪算法都不能保证得到最优解而沮丧，0/1背包问题是一个NP-复杂问题（见9.5.2节对NP-复杂问题的讨论）。对于这类问题，也许根本就不可能找到具有多项式时间的算法。虽然按  $p_i/w_i$  非递（增）减的次序装入物品不能保证得到最优解，但它是一个直觉上近似的解。我们希望它是一个好的启发式算法，且大多数时候能很好地接近最后算法。在600个随机产生的背包问题中，用这种启发式贪婪算法来解有239题为最优解。有583个例子与最优解相差10%，所有600个答案与最优解之差全在25%以内。该算法能在  $O(n \log n)$  时间内获得如此好的性能。

我们也许会问，是否存在一个  $x$  ( $x < 100$ )，使得贪婪启发法的结果与最优值相差在  $x\%$  以内。答案是否定的。为说明这一点，考虑例子  $n=2$ ,  $w=[1,y]$ ,  $p=[10,9y]$ , 和  $c=y$ 。贪婪算法结果为  $x=[1,0]$ ，这种方案的值为10。对于  $y = 10/9$ ，最优解的值为  $9y$ 。因此，贪婪算法的值与最优解的差对最优解的比例为  $((9y-10)/9y) \times 100\%$ ，对于大的  $y$ ，这个值趋近于100%。

但是可以建立贪婪启发式方法来提供解，使解的结果与最优解的值之差在最优值的  $x\%$  ( $x < 100$ ) 之内。首先将最多  $k$  件物品放入背包，如果这  $k$  件物品重量大于  $c$ ，则放弃它。否则，剩余的容量用来考虑将剩余物品按  $p_i/w_i$  递减的顺序装入。通过考虑由启发法产生的解法中最多为  $k$  件物品的所有可能的子集来得到最优解。

例13-9 考虑  $n=4$ ,  $w=[2,4,6,7]$ ,  $p=[6,10,12,13]$ ,  $c=11$ 。当  $k=0$  时，背包按物品价值密度非递减顺序装入，首先将物品1放入背包，然后是物品2，背包剩下的容量为5个单元，剩下的物品没有一个合适的，因此解为  $x=[1,1,0,0]$ 。此解获得的值为16。

现在考虑  $k=1$  时的贪婪启发法。最初的子集为  $\{1\}, \{2\}, \{3\}, \{4\}$ 。子集  $\{1\}, \{2\}$  产生与  $k=0$  时相同的结果，考虑子集  $\{3\}$ ，置  $x_3$  为1。此时还剩5个单位的容量，按价值密度非递增顺序来考虑如何利用这5个单位的容量。首先考虑物品1，它适合，因此取  $x_1$  为1，这时仅剩下3个单位容量



了，且剩余物品没有能够加入背包中的物品。通过子集 {3} 开始求解得结果为  $x=[1,0,1,0]$ ，获得的价值为 18。若从子集 {4} 开始，产生的解为  $x=[1,0,0,1]$ ，获得的价值为 19。考虑子集大小为 0 和 1 时获得的最优解为  $[1,0,0,1]$ 。这个解是通过  $k=1$  的贪婪启发式算法得到的。

若  $k=2$ ，除了考虑  $k<2$  的子集，还必需考虑子集  $\{1,2\}, \{1,3\}, \{1,4\}, \{2,3\}, \{2,4\}$  和  $\{3,4\}$ 。首先从最后一个子集开始，它是不可行的，故将其抛弃，剩下的子集经求解分别得到如下结果： $[1,1,0,0], [1,0,1,0], [1,0,0,1], [0,1,1,0]$  和  $[0,1,0,1]$ ，这些结果中最后一个价值为 23，它的值比  $k=0$  和  $k=1$  时获得的解要高，这个答案即为启发式方法产生的结果。

这种修改后的贪婪启发方法称为  $k$  阶优化方法 ( $k$ -optimal)。也就是，若从答案中取出  $k$  件物品，并放入另外  $k$  件，获得的结果不会比原来的好，而且用这种方式获得的值在最优值的  $(100/(k+1))\%$  以内。当  $k=1$  时，保证最终结果在最佳值的 50% 以内；当  $k=2$  时，则在 33.33% 以内等等，这种启发式方法的执行时间随  $k$  的增大而增加，需要测试的子集数目为  $O(n^k)$ ，每一个子集所需时间为  $O(n)$ ，因此当  $k>0$  时总的时间开销为  $O(n^{k+1})$ 。

实际观察到的性能要好得多，图 13-3 给出了 600 种随机测试的统计结果。

k	偏差百分比				
	0	1%	5%	10%	25%
0	239	390	528	583	600
1	360	527	598	600	
2	483	581	600		

图 13-3 600 个例子中差值在  $x\%$  以内的数目

### 13.3.3 拓扑排序

一个复杂的工程通常可以分解成一组小任务的集合，完成这些小任务意味着整个工程的完成。例如，汽车装配工程可分解为以下任务：将底盘放上装配线，装轴，将座位装在底盘上，上漆，装刹车，装门等等。任务之间具有先后关系，例如在装轴之前必须先将底板放上装配线。任务的先后顺序可用有向图表示——称为顶点活动 (Activity On Vertex, AOV) 网络。有向图的顶点代表任务，有向边  $(i, j)$  表示先后关系：任务  $j$  开始前任务  $i$  必须完成。图 13-4 显示了六个任务的工程，边  $(1, 4)$  表示任务 1 在任务 4 开始前完成，同样边  $(4, 6)$  表示任务 4 在任务 6 开始前完成，边  $(1, 4)$  与  $(4, 6)$  合起来可知任务 1 在任务 6 开始前完成，即前后关系是传递的。由此可知，边  $(1, 4)$  是多余的，因为边  $(1, 3)$  和  $(3, 4)$  已暗示了这种关系。

在很多条件下，任务的执行是连续进行的，例如汽车装配问题或平时购买的标有“需要装配”的消费品 (自行车、小孩的秋千装置，割草机等等)。我们可根据所建议的顺序来装配。在由任务建立的有向图中，边  $(i, j)$  表示在装配序列中任务  $i$  在任务  $j$  的前面，具有这种性质的序列称为拓扑序列 (topological orders 或 topological sequences)。根据任务的有向图建立拓扑序列的过程称为拓扑排序 (topological sorting)。

图 13-4 的任务有向图有多种拓扑序列，其中的三种为 123456，132456 和 215346，序列 142356 就不是拓扑序列，因为在这个序列中任务 4 在 3 的前面，而任务有向图中的边为  $(3, 4)$ ，这种序列与边  $(3, 4)$  及其他边所指示的序列相矛盾。

可用贪婪算法来建立拓扑序列。算法按从左到右的步骤构造拓扑序列，每一步在排好的序

列中加入一个顶点。利用如下贪婪准则来选择顶点：从剩下的顶点中，选择顶点  $w$ ，使得  $w$  不存在这样的入边  $(v, w)$ ，其中顶点  $v$  不在已排好的序列结构中出现。注意到如果加入的顶点  $w$  违背了这个准则（即有向图中存在边  $(v, w)$  且  $v$  不在已构造的序列中），则无法完成拓扑排序，因为顶点  $v$  必须跟随在顶点  $w$  之后。贪婪算法的伪代码如图 13-5 所示。while 循环的每次迭代代表贪婪算法的一个步骤。

现在用贪婪算法来求解图 13-4 的有向图。首先从一个空序列  $V$  开始，第一步选择  $V$  的第一个顶点。此时，在有向图中有两个候选顶点 1 和 2，若选择顶点 2，则序列  $V=2$ ，第一步完成。第二步选择  $V$  的第二个顶点，根据贪婪准则可知候选顶点为 1 和 5，若选择 5，则  $V=25$ 。下一步，顶点 1 是唯一的候选，因此  $V=251$ 。第四步，顶点 3 是唯一的候选，因此把顶点 3 加入  $V$  得到  $V=2513$ 。在最后两步分别加入顶点 4 和 6，得  $V=251346$ 。

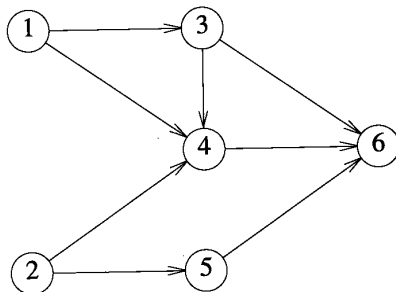


图13-4 任务有向图

```

设  $n$  是有向图中的顶点数
设  $V$  是一个空序列
while(true) {
    设  $w$  不存在入边  $(v, w)$ ，其中顶点  $v$  不在  $V$  中
    如果没有这样的  $w$ ，break。
    把  $w$  添加到  $V$  的尾部
}
if( $V$  中的顶点数少于  $n$ ) 算法失败
else  $V$  是一个拓扑序列

```

图13-5 拓扑排序

### 1. 贪婪算法的正确性

为保证贪婪算法的正确性，需要证明：1) 当算法失败时，有向图没有拓扑序列；2) 若算法没有失败， $V$  即是拓扑序列。2) 即是用贪婪准则来选取下一个顶点的直接结果，1) 的证明见定理 13-2，它证明了若算法失败，则有向图中有环路。若有向图中包含环  $q_j q_{j+1} \dots q_k q_j$ ，则它没有拓扑序列，因为该序列暗示了  $q_j$  一定要在  $q_j$  开始前完成。

定理 13-2 如果图 13-5 算法失败，则有向图含有环路。

证明 注意到当失败时  $|V| < n$ ，且没有候选顶点能加入  $V$  中，因此至少有一个顶点  $q_1$  不在  $V$  中，有向图中必包含边  $(q_2, q_1)$  且  $q_2$  不在  $V$  中，否则， $q_1$  是可加入  $V$  的候选顶点。同样，必有边  $(q_3, q_2)$  使得  $q_3$  不在  $V$  中，若  $q_3 = q_1$  则  $q_1 q_2 q_3$  是有向图中的一个环路；若  $q_3 \neq q_1$ ，则必存在  $q_4$  使  $(q_4, q_3)$  是有向图的边且  $q_4$  不在  $V$  中，否则， $q_3$  便是  $V$  的一个候选顶点。若  $q_4$  为  $q_1, q_2, q_3$  中的任何一个，则又可知有向图含有环，因为有向图具有有限个顶点数  $n$ ，继续利用上述方法，最后总能找到一个环路。



## 2. 数据结构的选择

为将图13-5用C++代码来实现,必须考虑序列V的描述方法,以及如何找出可加入V的候选顶点。一种高效的实现方法是将序列V用一维数组 $v$ 来描述的,用一个栈来保存可加入V的候选顶点。另有一个一维数组InDegree, InDegree[ $j$ ]表示与顶点 $j$ 相连的节点 $i$ 的数目,其中顶点 $i$ 不是V中的成员,它们之间的有向图的边表示为 $(i, j)$ 。当InDegree[ $j$ ]变为0时表示 $j$ 成为一个候选节点。序列V初始时空。InDegree[ $j$ ]为顶点 $j$ 的入度。每次向V中加入一个顶点时,所有与新加入顶点邻接的顶点 $j$ ,其InDegree[ $j$ ]减1。

对于有向图13-4,开始时InDegree[1:6]=[0,0,1,3,1,3]。由于顶点1和2的InDegree值为0,因此它们是可加入V的候选顶点,由此,顶点1和2首先入栈。每一步,从栈中取出一个顶点将其加入V,同时减去与其邻接的顶点的InDegree值。若在第一步时从栈中取出顶点2并将其加入V,便得到了 $v[0]=2$ ,和InDegree[1:6]=[0,0,1,2,0,3]。由于InDegree[5]刚刚变为0,因此将顶点5入栈。

程序13-2给出了相应的C++代码,这个代码被定义为Network的一个成员函数。而且,它对于有无加权的有向图均适用。但若用于无向图(不论其有无加权)将会得到错误的结果,因为拓扑排序是针对有向图来定义的。为解决这个问题,利用同样的模板来定义成员函数AdjacencyGraph, AdjacencyWGraph, LinkedGraph和LinkedWGraph。这些函数可重载Network中的函数并可输出错误信息。如果找到拓扑序列,则Topological函数返回true;若输入的有向图无拓扑序列则返回false。当找到拓扑序列时,将其返回到 $v[0:n-1]$ 中。

## 3. Network:Topological 的复杂性

第一和第三个for循环的时间开销为 $\Theta(n)$ 。若使用(耗费)邻接矩阵,则第二个for循环所用的时间为 $\Theta(n^2)$ ;若使用邻接链表,则所用时间为 $\Theta(n+e)$ 。在两个嵌套的while循环中,外层循环需执行 $n$ 次,每次将顶点 $w$ 加入到 $v$ 中,并初始化内层while循环。使用邻接矩阵时,内层while循环对于每个顶点 $w$ 需花费 $\Theta(n)$ 的时间;若利用邻接链表,则这个循环需花费 $d_w^{out}$ 的时间,因此,内层while循环的时间开销为 $\Theta(n^2)$ 或 $\Theta(n+e)$ 。所以,若利用邻接矩阵,程序13-2的时间复杂性为 $\Theta(n^2)$ ,若利用邻接链表则为 $\Theta(n+e)$ 。

程序13-2 拓扑排序

```
bool Network::Topological(int v[])
// 计算有向图中顶点的拓扑次序
// 如果找到了一个拓扑次序,则返回 true,此时,在v[0:n-1]中记录拓扑次序
// 如果不存在拓扑次序,则返回 false

int n = Vertices();

// 计算入度
int *InDegree = new int [n+1];
InitializePos(); // 图遍历器数组
for (int i = 1; i <= n; i++) // 初始化
    InDegree[i] = 0;
for (i = 1; i <= n; i++) // 从i出发的边
    int u = Begin(i);
    while (u) {
        InDegree[u]++;
```

```

    u = NextVertex(i);
}

// 把入度为 0 的顶点压入堆栈
LinkedStack<int> S;
for (i = 1; i <= n; i++)
    if (!InDegree[i]) S.Add(i);

// 产生拓扑次序
i = 0; // 数组 v 的游标
while (!S.IsEmpty()) { // 从堆栈中选择
    int w; // 下一个顶点
    S.Delete(w);
    v[i++] = w;
    int u = Begin(w);
    while (u) { // 修改入度
        InDegree[u]--;
        if (!InDegree[u]) S.Add(u);
        u = NextVertex(w);
    }
}

DeactivatePos();
delete [] InDegree;
return (i == n);
}

```

### 13.3.4 二分覆盖

二分图（见例 12-3）是一个无向图，它的  $n$  个顶点可二分为集合  $A$  和集合  $B$ ，且同一集合中的任意两个顶点在图中无边相连（即任何一条边都是一个顶点在集合  $A$  中，另一个在集合  $B$  中）。当且仅当  $B$  中的每个顶点至少与  $A$  中一个顶点相连时， $A$  的一个子集  $A'$  覆盖集合  $B$ （或简单地说， $A'$  是一个覆盖）。覆盖  $A'$  的大小即为  $A'$  中的顶点数目。当且仅当  $A'$  是覆盖  $B$  的子集中最小的时， $A'$  为最小覆盖。

例 13-10 考察如图 13-6 所示的具有 17 个顶点的二分图， $A=\{1, 2, 3, 16, 17\}$  和  $B=\{4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$ ，子集  $A'=\{1, 16, 17\}$  是  $B$  的最小覆盖。

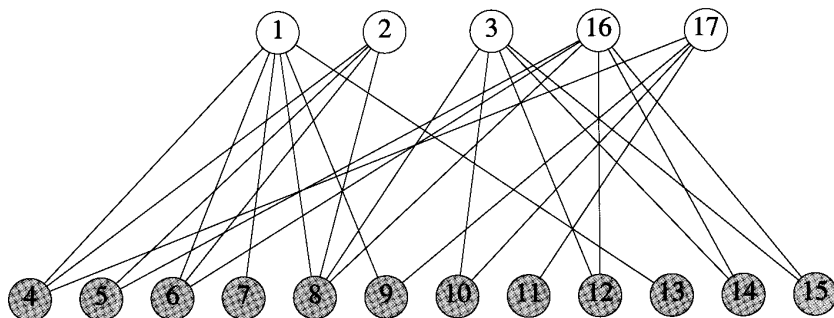


图 13-6 例 13-10 所使用的图

在二分图中寻找最小覆盖的问题为二分覆盖 (bipartite-cover) 问题。在例 12-3 中说明了最小覆盖是很有用的, 因为它能解决“在会议中使用最少的翻译人员进行翻译”这一类的问题。二分覆盖问题类似于集合覆盖 (set-cover) 问题。在集合覆盖问题中给出了  $k$  个集合  $S = \{S_1, S_2, \dots, S_k\}$ , 每个集合  $S_i$  中的元素均是全集  $U$  中的成员。当且仅当  $\bigcup_{i \in S'} S_i = U$  时,  $S$  的子集  $S'$  覆盖  $U$ ,  $S'$  中的集合数目即为覆盖的大小。当且仅当没有能覆盖  $U$  的更小的集合时, 称  $S'$  为最小覆盖。可以将集合覆盖问题转化为二分覆盖问题 (反之亦然), 即用  $A$  的顶点来表示  $S_1, \dots, S_k$ ,  $B$  中的顶点代表  $U$  中的元素。当且仅当  $S$  的相应集合中包含  $U$  中的对应元素时, 在  $A$  与  $B$  的顶点之间存在一条边。

例 13-11 令  $S = \{S_1, \dots, S_5\}$ ,  $U = \{4, 5, \dots, 15\}$ ,  $S_1 = \{4, 6, 7, 8, 9, 13\}$ ,  $S_2 = \{4, 5, 6, 8\}$ ,  $S_3 = \{8, 10, 12, 14, 15\}$ ,  $S_4 = \{5, 6, 8, 12, 14, 15\}$ ,  $S_5 = \{4, 9, 10, 11\}$ 。  $S' = \{S_1, S_4, S_5\}$  是一个大小为 3 的覆盖, 没有更小的覆盖,  $S'$  即为最小覆盖。这个集合覆盖问题可映射为图 13-6 的二分图, 即用顶点 1, 2, 3, 16 和 17 分别表示集合  $S_1, S_2, S_3, S_4$  和  $S_5$ , 顶点  $j$  表示集合中的元素  $j, 4 \leq j \leq 15$ 。

集合覆盖问题为 NP-复杂问题。由于集合覆盖与二分覆盖是同一类问题, 二分覆盖问题也是 NP-复杂问题。因此可能无法找到一个快速的算法来解决它, 但是可以利用贪婪算法寻找一种快速启发式方法。一种可能是分步建立覆盖  $A'$ , 每一步选择  $A$  中的一个顶点加入覆盖。顶点的选择利用贪婪准则: 从  $A$  中选取能覆盖  $B$  中还未被覆盖的元素数目最多的顶点。

例 13-12 考察图 13-6 所示的二分图, 初始化  $A' = \phi$  且  $B$  中没有顶点被覆盖, 顶点 1 和 16 均能覆盖  $B$  中的六个顶点, 顶点 3 覆盖五个, 顶点 2 和 17 分别覆盖四个。因此, 在第一步往  $A'$  中加入顶点 1 或 16, 若加入顶点 16, 则它覆盖的顶点为  $\{5, 6, 8, 12, 14, 15\}$ , 未覆盖的顶点为  $\{4, 7, 9, 10, 11, 13\}$ 。顶点 1 能覆盖其中四个顶点 ( $\{4, 7, 9, 13\}$ ), 顶点 2 覆盖一个 ( $\{4\}$ ), 顶点 3 覆盖一个 ( $\{10\}$ ), 顶点 16 覆盖零个, 顶点 17 覆盖四个  $\{4, 9, 10, 11\}$ 。下一步可选择 1 或 17 加入  $A'$ 。若选择顶点 1, 则顶点  $\{10, 11\}$  仍然未被覆盖, 此时顶点 1, 2, 16 不覆盖其中任意一个, 顶点 3 覆盖一个, 顶点 17 覆盖两个, 因此选择顶点 17, 至此所有顶点已被覆盖, 得  $A' = \{16, 1, 17\}$ 。

图 13-7 给出了贪婪覆盖启发式方法的伪代码, 可以证明: 1) 当且仅当初始的二分图没有覆盖时, 算法找不到覆盖; 2) 启发式方法可能找不到二分图的最小覆盖。

```

A' = φ
while (更多的顶点可被覆盖)
    把能覆盖未被覆盖的顶点数目最多的顶点加入 A'
if (有些顶点未被覆盖) 失败
else (找到一个覆盖)

```

图 13-7 贪婪覆盖启发式方法的伪代码

### 1. 数据结构的选取及复杂性分析

为实现图 13-7 的算法, 需要选择  $A'$  的描述方法及考虑如何记录  $A$  中节点所能覆盖的  $B$  中未覆盖节点的数目。由于对集合  $A'$  仅使用加法运算, 则可用一维整型数组  $C$  来描述  $A'$ , 用  $m$  来记录  $A'$  中元素个数。将  $A'$  中的成员记录在  $C[0:m-1]$  中。

对于  $A$  中顶点  $i$ , 令  $New_i$  为  $i$  所能覆盖的  $B$  中未覆盖的顶点数目。逐步选择  $New_i$  值最大的顶

点。由于一些原来未被覆盖的顶点现在被覆盖了，因此还要修改各  $New_i$  值。在这种更新中，检查  $B$  中最近一次被  $V$  覆盖的顶点，令  $j$  为这样的顶点，则  $A$  中所有覆盖  $j$  的顶点的  $New_i$  值均减1。

例13-13 考察图13-6，初始时  $(New_1, New_2, New_3, New_{16}, New_{17}) = (6, 4, 5, 6, 4)$ 。假设在例13-12中，第一步选择顶点16，为更新  $New_i$  的值检查  $B$  中所有最近被覆盖的顶点，这些顶点为5, 6, 8, 12, 14和15。当检查顶点5时，将顶点2和16的  $New_i$  值分别减1，因为顶点5不再是被顶点2和16覆盖的未覆盖节点；当检查顶点6时，顶点1, 2, 和16的相应值分别减1；同样，检查顶点8时，1, 2, 3和16的值分别减1；当检查完所有最近被覆盖的顶点，得到的  $New_i$  值为  $(4, 1, 0, 4)$ 。下一步选择顶点1，最新被覆盖的顶点为4, 7, 9和13；检查顶点4时， $New_1, New_2$ , 和  $New_{17}$  的值减1；检查顶点7时， $New_1$  的值减1，因为顶点1是覆盖7的唯一顶点。

为了实现顶点选取的过程，需要知道  $New_i$  的值及已被覆盖的顶点。可利用一个二维数组来达到这个目的， $New$  是一个整型数组， $New[i]$  即等于  $New_i$ ，且  $cov$  为一个布尔数组。若顶点  $i$  未被覆盖则  $cov[i]$  等于  $false$ ，否则  $cov[i]$  为  $true$ 。现将图13-7的伪代码进行细化得到图13-8。

```

m=0; //当前覆盖的大小
对于A中的所有i, New[i]=Degree[i]
对于B中的所有i, Cov[i]=false
while (对于A中的某些i, New[i]>0) {
    设v是具有最大的New[i]的顶点;
    C[m++]=v;
    for (所有邻接于v的顶点j) {
        if (!Cov[j]) {
            Cov[j]= true;
            对于所有邻接于j的顶点, 使其New[k]减1
        }
    }
}
if (有些顶点未被覆盖) 失败
else 找到一个覆盖

```

图13-8 图13-7的细化

更新  $New$  的时间为  $O(e)$ ，其中  $e$  为二分图中边的数目。若使用邻接矩阵，则需花  $\Theta(n^2)$  的时间来寻找图中的边，若用邻接链表，则需  $\Theta(n+e)$  的时间。实际更新时间根据描述方法的不同为  $O(n^2)$  或  $O(n+e)$ 。

逐步选择顶点所需时间为  $\Theta(\text{SizeOf}A)$ ，其中  $\text{SizeOf}A = |A|$ 。因为  $A$  的所有顶点都有可能被选择，因此所需步骤数为  $O(\text{SizeOf}A)$ ，覆盖算法总的复杂性为  $O(\text{SizeOf}A^2 + n^2) = O(n^2)$  或  $O(\text{SizeOf}A^2 + n + e)$ 。

## 2. 降低复杂性

通过使用有序数组  $New_i$ 、最大堆或最大选择树 (max selection tree) 可将每步选取顶点  $v$  的复杂性降为  $\Theta(1)$ 。但利用有序数组，在每步的最后需对  $New_i$  值进行重新排序。若使用箱子排序，则这种排序所需时间为  $\Theta(\text{SizeOf}B)$  ( $\text{SizeOf}B = |B|$ ) (见3.8.1节箱子排序)。由于一般  $\text{SizeOf}B$  比  $\text{SizeOf}A$  大得多，因此有序数组并不总能提高性能。

如果利用最大堆，则每一步都需要重建堆来记录 New 值的变化，可以在每次 New 值减 1 时进行重建。这种减法操作可引起被减的 New 值最多在堆中向下移一层，因此这种重建对于每次 New 值减 1 需  $\Theta(1)$  的时间，总共的减操作数目为  $O(e)$ 。因此在算法的所有步骤中，维持最大堆仅需  $O(e)$  的时间，因而利用最大堆时覆盖算法的总复杂性为  $O(n^2)$  或  $O(n+e)$ 。

若利用最大选择树，每次更新 New 值时需要重建选择树，所需时间为  $\Theta(\log \text{SizeOfA})$ 。重建的最好时机是在每步结束时，而不是在每次 New 值减 1 时，需要重建的次数为  $O(e)$ ，因此总的重建时间为  $O(e \log \text{SizeOfA})$ ，这个时间比最大堆的重建时间长一些。

然而，通过维持具有相同 New 值的顶点箱子，也可获得和利用最大堆时相同的时间限制。由于 New 的取值范围为 0 到  $\text{SizeOfB}$ ，需要  $\text{SizeOfB}+1$  个箱子，箱子  $i$  是一个双向链表，链接所有 New 值为  $i$  的顶点。在某一步结束时，假如 New[6] 从 12 变到 4，则需要将它从第 12 个箱子移到第 4 个箱子。利用模拟指针及一个节点数组 `node`（其中 `node[i]` 代表顶点  $i$ ，`node[i].left` 和 `node[i].right` 为双向链表指针），可将顶点 6 从第 12 个箱子移到第 4 个箱子，从第 12 个箱子中删除 `node[6]` 并将其插入第 4 个箱子。利用这种箱子模式，可得覆盖启发式算法的复杂性为  $O(n^2)$  或  $O(n+e)$ 。（取决于利用邻接矩阵还是线性表来描述图）。

### 3. 双向链接箱子的实现

为了实现上述双向链接箱子，图 13-9 定义了类 `Undirected` 的私有成员。`NodeType` 是一个具有私有整型成员 `left` 和 `right` 的类，它的数据类型是双向链表节点，程序 13-3 给出了 `Undirected` 的私有成员的代码。

```
void CreateBins (int b, int n)
    创建b个空箱子和n个节点

void DestroyBins() { delete [] node;
                    delete [] bin;}

void InsertBins(int b, int v)
    在箱子b中添加顶点v

void MoveBins(int bMax, int ToBin, int v)
    从当前箱子中移动顶点v到箱子ToBin

int *bin;
    bin[i]指向代表该箱子的双向链表的首节点

NodeType *node;
    node[i]代表存储顶点i的节点
```

图 13-9 实现双向链接箱子所需的 `Undirected` 私有成员

程序 13-3 箱子函数的定义

```
void Undirected::CreateBins(int b, int n)
{// 创建b个空箱子和n个节点
    node = new NodeType [n+1];
```

```

    bin = new int [b+1];
    // 将箱子置空
    for (int i = 1; i <= b; i++)
        bin[i] = 0;
}

void Undirected::InsertBins(int b, int v)
{// 若b不为0, 则将 v 插入箱子 b
    if (!b) return; // b为0, 不插入
    node[v].left = b; // 添加在左端
    if (bin[b]) node[bin[b]].left = v;
    node[v].right = bin[b];
    bin[b] = v;
}

void Undirected::MoveBins(int bMax, int ToBin, int v)
{// 将顶点 v 从其当前所在箱子移动到 ToBin.
    // v的左、右节点
    int l = node[v].left;
    int r = node[v].right;

    // 从当前箱子中删除
    if (r) node[r].left = node[v].left;
    if (l > bMax || bin[l] != v) // 不是最左节点
        node[l].right = r;
    else bin[l] = r; // 箱子 l的最左边

    // 添加到箱子 ToBin
    InsertBins(ToBin, v);
}

```

函数CreateBins动态分配两个数组：node和bin，node[i]表示顶点i，bin[i]指向其New值为i的双向链表的顶点，for循环将所有双向链表置为空。

如果 $b = 0$ ，函数InsertBins 将顶点v 插入箱子b 中。因为b 是顶点v 的New 值， $b=0$ 意味着顶点v 不能覆盖B中当前还未被覆盖的任何顶点，所以，在建立覆盖时这个箱子没有用处，故可以将其舍去。当 $b \neq 0$ 时，顶点n 加入New 值为b 的双向链表箱子的最前面，这种加入方式需要将node[v] 加入bin[b] 中第一个节点的左边。由于表的最左节点应指向它所属的箱子，因此将它的node[v].left 置为b。若箱子不空，则当前第一个节点的 left 指针被置为指向新节点。node[v] 的右指针被置为bin[b]，其值可能为0或指向上一个首节点的指针。最后，bin[b]被更新为指向表中新的第一个节点。

MoveBins 将顶点v 从它在双向链表中的当前位置移到 New 值为ToBin 的位置上。其中存在bMax，使得对所有的箱子bin[j]都有：如 $j > bMax$ ，则bin[j]为空。代码首先确定node[v]在当前双向链表中的左右节点，接着从双链表中取出 node[v]，并利用InsertBins函数将其重新插入到bin[ToBin]中。

#### 4. Undirected::BipartiteCover的实现

函数的输入参数L用于分配图中的顶点（分配到集合A或B）。 $L[i]=1$ 表示顶点i在集合A中， $L[i]=2$ 则表示顶点在B中。函数有两个输出参数：C和m，m为所建立的覆盖的大小， $C[0,m-1]$

是A中形成覆盖的顶点。若二分图没有覆盖，函数返回 false；否则返回 true。完整的代码见程序13-4。

程序13-4 构造贪婪覆盖

```
bool Undirected::BipartiteCover(int L[], int C[], int& m)
{// 寻找一个二分图覆盖
// L 是输入顶点的标号, L[i] = 1 当且仅当 i 在 A 中
// C 是一个记录覆盖的输出数组
// 如果图中不存在覆盖, 则返回 false
// 如果图中有一个覆盖, 则返回 true;
// 在 m 中返回覆盖的大小; 在 C[0:m-1] 中返回覆盖

int n = Vertices();

// 插件结构
int SizeOfA = 0;
for (int i = 1; i <= n; i++) // 确定集合 A 的大小
    if (L[i] == 1) SizeOfA++;
int SizeOfB = n - SizeOfA;
CreateBins(SizeOfB, n);
int *New = new int [n+1]; // 顶点 i 覆盖了 B 中 New[i] 个未被覆盖的顶点
bool *Change = new bool [n+1]; // Change[i] 为 true 当且仅当 New[i] 已改变
bool *Cov = new bool [n+1]; // Cov[i] 为 true 当且仅当顶点 i 被覆盖
InitializePos();
LinkedList<int> S;

// 初始化
for (i = 1; i <= n; i++) {
    Cov[i] = Change[i] = false;
    if (L[i] == 1) { // i 在 A 中
        New[i] = Degree(i); // i 覆盖了这么多
        InsertBins(New[i], i);}

// 构造覆盖
int covered = 0, // 被覆盖的顶点
    MaxBin = SizeOfB; // 可能非空的箱子
m = 0; // C 的游标
while (MaxBin > 0) { // 搜索所有箱子
    // 选择一个顶点
    if (bin[MaxBin]) { // 箱子不空
        int v = bin[MaxBin]; // 第一个顶点
        C[m++] = v; // 把 v 加入覆盖
        // 标记新覆盖的顶点
        int j = Begin(v), k;
        while (j) {
            if (!Cov[j]) { // j 尚未被覆盖
```



```

    Cov[j] = true;
    covered++;
    // 修改 New
    k = Begin(j);
    while (k) {
        New[k]--; // j 不计入在内
        if (!Change[k]) {
            S.Add(k); // 仅入栈一次
            Change[k] = true;
        }
        k = NextVertex(j);
    }
    j = NextVertex(v);

    // 更新箱子
    while (!S.IsEmpty()) {
        S.Delete(k);
        Change[k] = false;
        MoveBins(SizeOfB, New[k], k);
    }
    else MaxBin--;
}

DeactivatePos();
DestroyBins();
delete [] New;
delete [] Change;
delete [] Cov;
return (covered == SizeOfB);
}

```

程序13-4首先计算出集合 $A$ 和 $B$ 的大小、初始化必要的双向链表结构、创建三个数组、初始化图遍历器、并创建一个栈。然后将数组  $Cov$  和  $Change$  初始化为  $false$ ，并将 $A$ 中的顶点根据它们覆盖 $B$ 中顶点的数目插入到相应的双向链表中。

为了构造覆盖，首先按  $SizeOfB$  递减至1的顺序检查双向链表。当发现一个非空的表时，就将其第一个顶点 $v$  加入到覆盖中，这种策略即为选择具有最大  $New$  值的顶点。将所选择的顶点加入覆盖数组  $C$  并检查 $B$ 中所有与它邻接的顶点。若顶点  $j$  与 $v$  邻接且还未被覆盖，则将  $Cov[j]$  置为  $true$ ，表示顶点 $j$  现在已被覆盖，同时将已被覆盖的 $B$ 中的顶点数目加1。由于 $j$  是最近被覆盖的，所有 $A$ 中与 $j$  邻接的顶点的  $New$  值减1。下一个  $while$  循环降低这些  $New$  值并将  $New$  值被降低的顶点保存在一个栈中。当所有与顶点 $v$  邻接的顶点的  $Cov$  值更新完毕后， $New$  值反映了 $A$ 中每个顶点所能覆盖的新的顶点数，然而 $A$ 中的顶点由于  $New$  值被更新，处于错误的双向链表中，下一个  $while$  循环则将这些顶点移到正确的表中。

### 13.3.5 单源最短路径

在这个问题中，给出有向图  $G$ ，它的每条边都有一个非负的长度（耗费） $a[i][j]$ ，路径的长度即为此路径所经过的边的长度之和。对于给定的源顶点  $s$ ，需找出从它到图中其他任意顶点（称为目的）的最短路径。图13-10a 给出了一个具有五个顶点的有向图，各边上的数即为长度。假设源顶点 $s$  为1，从顶点1出发的最短路径按路径长度顺序列在图13-10b 中，每条路径前

面的数字为路径的长度。

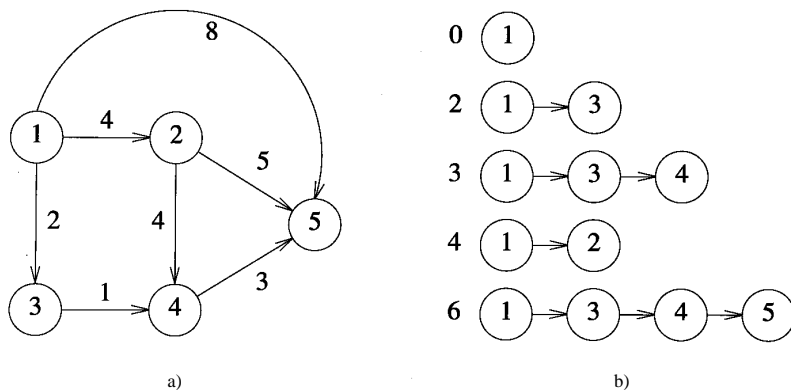


图13-10 最短路径举例

a) 图 b) 最短路径

利用E. Dijkstra发明的贪婪算法可以解决最短路径问题，它通过分步方法求出最短路径。每一步产生一个到达新的目的顶点的最短路径。下一步所能达到的目的顶点通过如下贪婪准则选取：在还未产生最短路径的顶点中，选择路径长度最短的目的顶点。也就是说，Dijkstra的方法按路径长度顺序产生最短路径。

首先最初产生从 $s$ 到它自身的路径，这条路径没有边，其长度为0。在贪婪算法的每一步中，产生下一个最短路径。一种方法是在目前已产生的最短路径中加入一条可行的最短的边，结果产生的新路径是原先产生的最短路径加上一条边。这种策略并不总是起作用。另一种方法是在目前产生的每一条最短路径中，考虑加入一条最短的边，再从所有这些边中先选择最短的，这种策略即是Dijkstra算法。

可以验证按长度顺序产生最短路径时，下一条最短路径总是由一条已产生的最短路径加上一条边形成。实际上，下一条最短路径总是由已产生的最短路径再扩充一条最短的边得到的，且这条路径所到达的顶点其最短路径还未产生。例如在图13-10中，b中第二条路径是第一条路径扩充一条边形成的；第三条路径则是第二条路径扩充一条边；第四条路径是第一条路径扩充一条边；第五条路径是第三条路径扩充一条边。

通过上述观察可用一种简便的方法来存储最短路径。可以利用数组 $p$ ， $p[i]$ 给出从 $s$ 到达 $i$ 的路径中顶点 $i$ 前面的那个顶点。在本例中 $p[1:5]=[0,1,1,3,4]$ 。从 $s$ 到顶点 $i$ 的路径可反向创建。从 $i$ 出发按 $p[i], p[p[i]], p[p[p[i]]], \dots$ 的顺序，直到到达顶点 $s$ 或0。在本例中，如果从 $i=5$ 开始，则顶点序列为 $p[i]=4, p[4]=3, p[3]=1=s$ ，因此路径为1,3,4,5。

为能方便地按长度递增的顺序产生最短路径，定义 $d[i]$ 为在已产生的最短路径中加入一条最短边的长度，从而使得扩充的路径到达顶点 $i$ 。最初，仅有从 $s$ 到 $s$ 的一条长度为0的路径，这时对于每个顶点 $i$ ， $d[i]$ 等于 $a[s][i]$ （ $a$ 是有向图的长度邻接矩阵）。为产生下一条路径，需要选择还未产生最短路径的下一个节点，在这些节点中 $d$ 值最小的即为下一条路径的终点。当获得一条新的最短路径后，由于新的最短路径可能会产生更小的 $d$ 值，因此有些顶点的 $d$ 值可能会发生变化。

综上所述，可以得到图13-11所示的伪代码，1) 将与 $s$ 邻接的所有顶点的 $p$ 初始化为 $s$ ，这个初始化用于记录当前可用的最好信息。也就是说，从 $s$ 到 $i$ 的最短路径，即是由 $s$ 到它自身那

条路径再扩充一条边得到。当找到更短的路径时， $p[i]$ 值将被更新。若产生了下一条最短路径，需要根据路径的扩充边来更新 $d$ 的值。

- 1) 初始化 $d[i]=a[s][i]$  ( $1 \leq i \leq n$ )，  
对于邻接于 $s$ 的所有顶点 $i$ ，置 $p[i]=s$ ，对于其余的顶点置 $p[i]=0$ ；  
对于 $p[i] \neq 0$ 的所有顶点建立 $L$ 表。
- 2) 若 $L$ 为空，终止，否则转至3)。
- 3) 从 $L$ 中删除 $d$ 值最小的顶点。
- 4) 对于与 $i$ 邻接的所有还未到达的顶点 $j$ ，更新 $d[j]$ 值为 $\min\{d[j], d[i]+a[i][j]\}$ ；若 $d[j]$ 发生了变化且 $j$ 还未在 $L$ 中，则置 $p[j]=i$ ，并将 $j$ 加入 $L$ ，转至2)。

图13-11 最短路径算法的描述

### 1. 数据结构的选择

我们需要为未到达的顶点列表 $L$ 选择一个数据结构。从 $L$ 中可以选出 $d$ 值最小的顶点。如果 $L$ 用最小堆（见9.3节）来维护，则这种选取可在对数时间内完成。由于3)的执行次数为 $O(n)$ ，所以所需时间为 $O(n \log n)$ 。由于扩充一条边产生新的最短路径时，可能使未到达的顶点产生更小的 $d$ 值，所以在4)中可能需要改变一些 $d$ 值。虽然算法中的减操作并不是标准的最小堆操作，但它能在对数时间内完成。由于执行减操作的总次数为： $O(\text{有向图中的边数}) = O(n^2)$ ，因此执行减操作的总时间为 $O(n^2 \log n)$ 。

若 $L$ 用无序的链表来维护，则3)与4)花费的时间为 $O(n^2)$ ，3)的每次执行需 $O(|L|) = O(n)$ 的时间，每次减操作需 $O(1)$ 的时间（需要减去 $d[j]$ 的值，但链表不用改变）。

利用无序链表将图13-11的伪代码细化为程序13-5，其中使用了Chain（见程序3-8）和ChainIterator类（见程序3-18）。

程序13-5 最短路径程序

```
template<class T>
void AdjacencyWDigraph<T>::ShortestPaths(int s, T d[], int p[])
// 寻找从顶点 s 出发的最短路径，在 d 中返回最短距离
// 在 p 中返回前继顶点
if (s < 1 || s > n) throw OutOfBounds();
Chain<int> L; // 路径可到达顶点的列表
ChainIterator<int> I;
// 初始化 d, p, L
for (int i = 1; i <= n; i++){
    d[i] = a[s][i];
    if (d[i] == NoEdge) p[i] = 0;
    else {p[i] = s;
        L.Insert(0,i);}
}

// 更新 d, p
while (!L.IsEmpty()) { // 寻找具有最小 d 的顶点 v
    int *v = I.Initialize(L);
```

```

int *w = l.Next();
while (w) {
    if (d[*w] < d[*v]) v = w;
    w = l.Next();}

// 从L中删除通向顶点v的下一最短路径并更新d
int i = *v;
L.Delete(*v);
for (int j = 1; j <= n; j++) {
    if (a[i][j] != NoEdge && (!p[j] ||
        d[j] > d[i] + a[i][j])) {
        // 减小d[j]
        d[j] = d[i] + a[i][j];
        // 将j加入L
        if (!p[j]) L.Insert(0,j);
        p[j] = i;}
    }
}
}

```

若NoEdge足够大，使得没有最短路径的长度大于或等于 NoEdge，则最后一个for 循环的if 条件可简化为：

```

if (d[j] > d[i] + a[i][j]))
NoEdge 的值应在能使d[j]+a[i][j] 不会产生溢出的范围内。

```

## 2. 复杂性分析

程序13-5的复杂性是 $O(n^2)$ ，任何最短路径算法必须至少对每条边检查一次，因为任何一条边都有可能在最短路径中。因此这种算法的最小可能时间为 $O(e)$ 。由于使用耗费邻接矩阵来描述图，仅决定哪条边在有向图中就需 $O(n^2)$ 的时间。因此，采用这种描述方法的算法需花费 $O(n^2)$ 的时间。不过程序13-5作了优化（常数因子级）。即使改变邻接表，也只会使最后一个for 循环的总时间降为 $O(e)$ （因为只有与i 邻接的顶点的d 值改变）。从L中选择及删除最小距离的顶点所需总时间仍然是 $O(n^2)$ 。

### 13.3.6 最小耗费生成树

在例12-2及13-3中已考察过这个问题。因为具有 $n$ 个顶点的无向网络 $G$ 的每个生成树刚好具有 $n-1$ 条边，所以问题是用某种方法选择 $n-1$ 条边使它们形成 $G$ 的最小生成树。至少可以采用三种不同的贪婪策略来选择这 $n-1$ 条边。这三种求解最小生成树的贪婪算法策略是：Kruskal算法，Prim算法和Sollin算法。

#### 1. Kruskal算法

##### (1) 算法思想

Kruskal算法每次选择 $n-1$ 条边，所使用的贪婪准则是：从剩下的边中选择一条不会产生环路的具有最小耗费的边加入已选择的边的集合中。注意到所选取的边若产生环路则不可能形成一棵生成树。Kruskal算法分 $e$ 步，其中 $e$ 是网络中边的数目。按耗费递增的顺序来考虑这 $e$ 条边，每次考虑一条边。当考虑某条边时，若将其加入到已选边的集合中会出现环路，则将其抛弃，否则，将它选入。

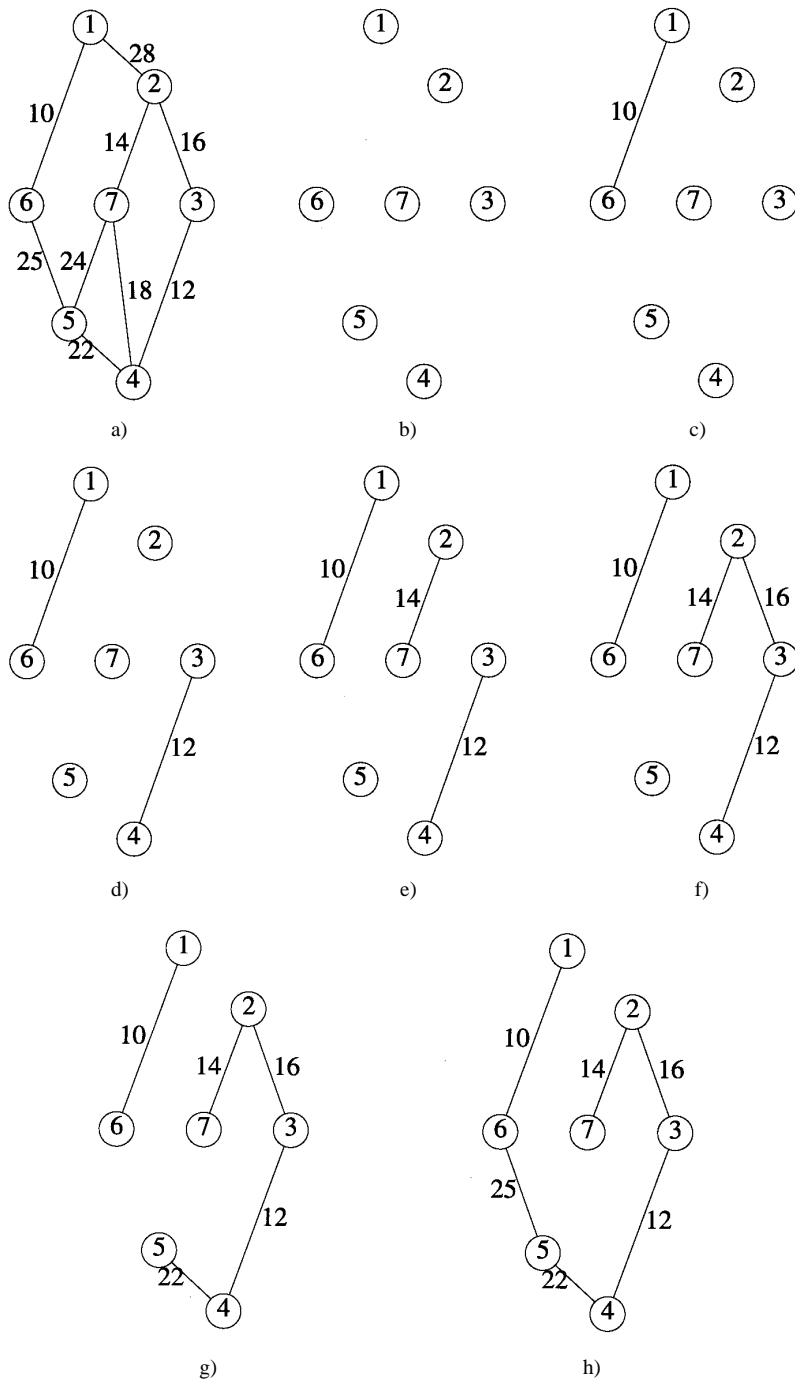


图13-12 构造最小耗费生成树

考察图 13-12a 中的网络。初始时没有任何边被选择。图 13-12b 显示了各节点的当前状态。边 (1,6) 是最先选入的边，它被加入到欲构建的生成树中，得到图 13-12c。下一步选择边 (3,4) 并将其加入树中 (如图 13-12d 所示)。然后考虑边 (2,7)，将它加入树中并不会产生环路，于是便得到图 13-12e。下一步考虑边 (2,3) 并将其加入树中 (如图 13-12f 所

示)。在其余还未考虑的边中,  $(7, 4)$  具有最小耗费, 因此先考虑它, 将它加入正在创建的树中会产生环路, 所以将其丢弃。此后将边  $(5, 4)$  加入树中, 得到的树如图 13-12g 所示。下一步考虑边  $(7, 5)$ , 由于会产生环路, 将其丢弃。最后考虑边  $(6, 5)$  并将其加入树中, 产生了一棵生成树 (如图 13-12h 所示), 其耗费为 99。图 13-13 给出了 Kruskal 算法的伪代码。

```
//在一个具有  $n$  个顶点的网络中找到一棵最小生成树
令  $T$  为所选边的集合, 初始化  $T = \phi$ 
令  $E$  为网络中边的集合
while( $E \neq \phi$  && ( $|T| < n-1$ )){
    令  $(u, v)$  为  $E$  中代价最小的边
     $E = E - \{(u, v)\}$  //从  $E$  中删除边
    if( $(u, v)$  加入  $T$  中不会产生环路) 将  $(u, v)$  加入  $T$ 
}
if( $|T| == n-1$ )  $T$  是最小耗费生成树
else 网络不是互连的, 不能找到生成树
```

图13-13 Kruskal算法的伪代码

## (2) 正确性证明

利用前述装载问题所用的转化技术可以证明图 13-13 的贪婪算法总能建立一棵最小耗费生成树。需要证明以下两点: 1) 只要存在生成树, Kruskal 算法总能产生一棵生成树; 2) 产生的生成树具有最小耗费。

令  $G$  为任意加权无向图 (即  $G$  是一个无向网络)。从 12.11.3 节可知当且仅当一个无向图连通时它有生成树。而且在 Kruskal 算法中被拒绝 (丢弃) 的边是那些会产生环路的边。删除连通图环路中的一条边所形成的图仍是连通图, 因此如果  $G$  在开始时是连通的, 则  $T$  与  $E$  中的边总能形成一个连通图。也就是若  $G$  开始时是连通的, 算法不会终止于  $E = \phi$  和  $|T| < n-1$ 。

现在来证明所建立的生成树  $T$  具有最小耗费。由于  $G$  具有有限棵生成树, 所以它至少具有一棵最小生成树。令  $U$  为这样的一棵最小生成树,  $T$  与  $U$  都刚好有  $n-1$  条边。如果  $T = U$ , 则  $T$  就具有最小耗费, 那么不必再证明下去。因此假设  $T \neq U$ , 令  $k (k > 0)$  为在  $T$  中而不在  $U$  中的边的个数, 当然  $k$  也是在  $U$  中而不在  $T$  中的边的数目。

通过把  $U$  变换为  $T$  来证明  $U$  与  $T$  具有相同的耗费, 这种转化可在  $k$  步内完成。每一步使在  $T$  而不在  $U$  中的边的数目刚好减 1。而且  $U$  的耗费不会因为转化而改变。经过  $k$  步的转化得到的  $U$  将与原来的  $U$  具有相同的耗费, 且转化后  $U$  中的边就是  $T$  中的边。由此可知,  $T$  具有最小耗费。

每步转化包括从  $T$  中移一条边  $e$  到  $U$  中, 并从  $U$  中移出一条边  $f$ 。边  $e$  与  $f$  的选取按如下方式进行:

1) 令  $e$  是在  $T$  中而不在  $U$  中的具有最小耗费的边。由于  $k > 0$ , 这条边肯定存在。

2) 当把  $e$  加入  $U$  时, 则会形成唯一的一条环路。令  $f$  为这条环路上不在  $T$  中的任意一条边。由于  $T$  中不含环路, 因此所形成的环路中至少有一条边不在  $T$  中。

从  $e$  与  $f$  的选择方法中可以看出,  $V = U + \{e\} - \{f\}$  是一棵生成树, 且  $T$  中恰有  $k-1$  条边不在  $V$  中出现。现在来证明  $V$  的耗费与  $U$  的相同。显然,  $V$  的耗费等于  $U$  的耗费加上边  $e$  的耗费再减去

边 $f$ 的耗费。若 $e$ 的耗费比 $f$ 的小,则生成树 $V$ 的耗费比 $U$ 的耗费小,这是不可能的。如果 $e$ 的耗费高于 $f$ ,在Kruskal算法中 $f$ 会在 $e$ 之前被考虑。由于 $f$ 不在 $T$ 中,Kruskal算法在考虑 $f$ 能否加入 $T$ 时已将 $f$ 丢弃,因此 $f$ 和 $T$ 中耗费小于或等于 $f$ 的边共同形成环路。通过选择 $e$ ,所有这些边均在 $U$ 中,因此 $U$ 肯定含有环路,但是实际上这不可能,因为 $U$ 是一棵生成树。 $e$ 的代价高于 $f$ 的假设将会导致矛盾。剩下的唯一的可能是 $e$ 与 $f$ 具有相同的耗费,由此可知 $V$ 与 $U$ 的耗费相同。

### (3) 数据结构的选择及复杂性分析

为了按耗费非递减的顺序选择边,可以建立最小堆并根据需要从堆中一条一条地取出各边。当图中有 $e$ 条边时,需花 $\Theta(e)$ 的时间初始化堆及 $O(\log e)$ 的时间来选取每一条边。

边的集合 $T$ 与 $G$ 中的顶点一起定义了一个由至多 $n$ 个连通子图构成的图。用顶点集合来描述每个子图,这些顶点集合没有公共顶点。为了确定边 $(u,v)$ 是否会产生环路,仅需检查 $u,v$ 是否在同一个顶点集中(即处于同一子图)。如果是,则会形成一个环路;如果不是,则不会产生环路。因此对于顶点集使用两个Find操作就足够了。当一条边包含在 $T$ 中时,2个子图将被合并成一个子图,即对两个集合执行Union操作。集合的Find和Union操作可利用8.10.2节的树(以及加权规则和路径压缩)来高效地执行。Find操作的次数最多为 $2e$ ,Union操作的次数最多为 $n-1$ (若网络是连通的,则刚好是 $n-1$ 次)。加上树的初始化时间,算法中这部分的复杂性只比 $O(n+e)$ 稍大一点。

对集合 $T$ 所执行的唯一操作是向 $T$ 中添加一条新边。 $T$ 可用数组 $t$ 来实现。添加操作在数组的一端进行,因为最多可在 $T$ 中加入 $n-1$ 条边,因此对 $T$ 的操作总时间为 $O(n)$ 。

总结上述各个部分的执行时间,可得图13-13算法的渐进复杂性为 $O(n+e \log e)$ 。

### (4) 实现

利用上述数据结构,图13-13可用C++代码来实现。首先定义EdgeNode类(见程序13-6),它是最小堆的元素及生成树数组 $t$ 的数据类型。

程序13-6 Kruskal算法所需要的数据类型

```
template <class T>
class EdgeNode {
public:
    operator T() const {return weight;}
private:
    T weight;//边的高度
    int u, v;//边的端点
};
```

为了更简单地使用8.10.2节的查找和合并策略,定义了UnionFind类,它的构造函数是程序8-16的初始化函数,Union是程序8-16的加权合并函数,Find是程序8-17的路径压缩搜索函数。

为了编写与网络描述无关的代码,还定义了一个新的类UNetWork,它包含了应用于无向网络的所有函数。这个类与Undirected类的差别在于Undirected类中的函数不要求加权边,而UNetWork要求边必须带有权值。UNetWork中的成员需要利用Network类中定义的诸如Begin和NextVertex的遍历函数。不过,新的遍历函数不仅需要返回下一个邻接的顶点,而且要返回到达这个顶点的边的权值。这些遍历函数以及有向和无向加权网络的其他函数一起构成了WNetwork类(见程序13-7)。



## 程序13-7 WNetwork类

---

```
template<class T>
class WNetwork : virtual public Network
{
public :
    virtual void First(int i, int& j, T& c)=0;
    virtual void Next(int i, int& j, T& c)=0;
};
```

---

象Begin和NextVertex一样，可在AdjacencyWDigraph及LinkedWDigraph类中加入函数First与Next。现在AdjacencyWDigraph及LinkedWDigraph类都需要从WNetWork中派生而来。由于AdjacencyWGraph类和LinkedWGraph类需要访问UNetwork的成员，所以这两个类还必须从UNetWork中派生而来。UNetWork::Kruskal的代码见程序13-8，它要求将Edges()定义为NetWork类的虚成员，并且把UNetWork定义为EdgeNode的友元)。如果没有生成树，函数返回false，否则返回true。注意当返回true时，在数组t中返回最小耗费生成树。

## 程序13-8 Kruskal算法的C++代码

---

```
template<class T>
bool UNetwork<T>::Kruskal(EdgeNode<T> t[])
{// 使用Kruskal算法寻找最小耗费生成树
// 如果不连通则返回false
// 如果连通，则在t[0:n-2]中返回最小生成树

int n = Vertices();
int e = Edges();
//设置网络边的数组
InitializePos(); // 图遍历器
EdgeNode<T> *E = new EdgeNode<T> [e+1];
int k = 0; // E的游标
for (int i = 1; i <= n; i++) { // 使所有边附属于 i
    int j;
    T c;
    First(i, j, c);
    while (j) { // j 邻接自 i
        if (i < j) { // 添加到达 E的边
            E[++k].weight = c;
            E[k].u = i;
            E[k].v = j;}
        Next(i, j, c);
    }
}

// 把边放入最小堆
MinHeap<EdgeNode<T> > H(1);
H.Initialize(E, e, e);

UnionFind U(n); // 合并/搜索结构
```

```

// 根据耗费的次序来抽取边
k = 0; // 此时作为 t 的游标
while (e && k < n - 1) {
    // 生成树未完成，尚有剩余边
    EdgeNode<T> x;
    H.DeleteMin(x); // 最小耗费边
    e--;
    int a = U.Find(x.u);
    int b = U.Find(x.v);
    if (a != b) { // 选择边
        t[k++] = x;
        U.Union(a,b);
    }

    DeactivatePos();
    H.Deactivate();
    return (k == n - 1);
}

```

## 2. Prim算法

与Kruskal算法类似，Prim算法通过每次选择多条边来创建最小生成树。选择下一条边的贪婪准则是：从剩余的边中，选择一条耗费最小的边，并且它的加入应使所有入选的边仍是一棵树。最终，在所有步骤中选择的边形成一棵树。相反，在Kruskal算法中所有入选的边集合最终形成一个森林。

Prim算法从具有一个单一顶点的树 $T$ 开始，这个顶点可以是原图中任意一个顶点。然后往 $T$ 中加入一条代价最小的边 $(u,v)$ 使 $T \cup \{(u,v)\}$ 仍是一棵树，这种加边的步骤反复循环直到 $T$ 中包含 $n-1$ 条边。注意对于边 $(u,v)$ ， $u$ 、 $v$ 中正好有一个顶点位于 $T$ 中。Prim算法的伪代码如图13-14所示。在伪代码中也包含了所输入的图不是连通图的可能，在这种情况下没有生成树。图13-15显示了对图13-12a使用Prim算法的过程。把图13-14的伪代码细化为C++程序及其正确性的证明留作练习（练习31）。

```

//假设网络中至少具有一个顶点
设 $T$ 为所选择的边的集合，初始化 $T=\phi$ 
设 $TV$ 为已在树中的顶点的集合，置 $TV=\{1\}$ 
令 $E$ 为网络中边的集合
while( $E \neq \phi$ ) && ( $|T| < n-1$ ) {
    令 $(u,v)$ 为最小代价边，其中 $u \in TV, v \notin TV$ 
    if (没有这种边) break
     $E = E - \{(u,v)\}$  //从 $E$ 中删除此边
    在 $T$ 中加入边 $(u,v)$ 
}
if ( $|T| == n-1$ )  $T$ 是一棵最小生成树
else 网络是不连通的，没有最小生成树

```

图13-14 Prim最小生成树算法

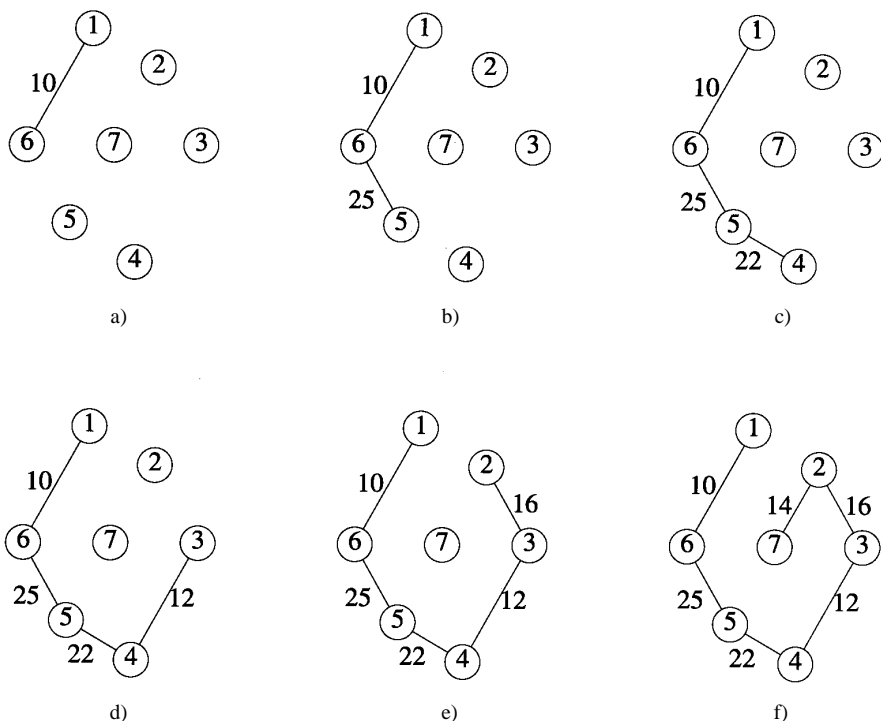


图13-15 Prim算法的步骤

如果根据每个不在 $TV$ 中的顶点 $v$ 选择一个顶点 $near(v)$ , 使得 $near(v) \in TV$ 且 $cost(v, near(v))$ 的值是所有这样的 $near(v)$ 节点中最小的, 则实现Prim算法的时间复杂性为 $O(n^2)$ 。下一条添加到 $T$ 中的边是这样的边: 其 $cost(v, near(v))$ 最小, 且 $v \notin TV$ 。

### 3. Sollin算法

Sollin算法每步选择若干条边。在每步开始时, 所选择的边及图中的 $n$ 个顶点形成一个生成树的森林。在每一步中为森林中的每棵树选择一条边, 这条边刚好有一个顶点在树中且边的代价最小。将所选择的边加入要创建的生成树中。注意一个森林中的两棵树可选择同一条边, 因此必须多次复制同一条边。当有多条边具有相同的耗费时, 两棵树可选择与它们相连的不同的边, 在这种情况下, 必须丢弃其中的一条边。开始时, 所选择的边的集合为空。若某一步结束时仅剩下一棵树或没有剩余的边可供选择时算法终止。

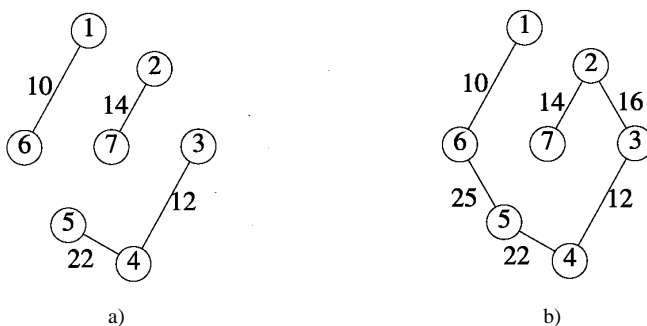


图13-16 Sollin算法的步骤

图13-6给出了初始状态为图13-12a时,使用Sollin算法的步骤。初始入选边数为0时的情形如图13-12a时,森林中的每棵树均是单个顶点。顶点1,2,...,7所选择的边分别是(1,6), (2,7), (3,4), (4,3), (5,4), (6,1), (7,2), 其中不同的边是(1,6), (2,7), (3,4)和(5,4), 将这些边加入入选边的集合后所得到的结果如图13-16a所示。下一步具有顶点集{1,6}的树选择边(6,5), 剩下的两棵树选择边(2,3), 加入这两条边后已形成一棵生成树, 构建好的生成树见图13-6b。Sollin算法的C++程序实现及其正确性证明留作练习(练习32)。

## 练习

8. 针对装载问题, 扩充贪婪算法, 考虑有两条船的情况, 算法总能产生最优解吗?

9. 已知 $n$ 个任务的执行序列。假设任务 $i$ 需要 $t_i$ 个时间单位。若任务完成的顺序为 $1, 2, \dots, n$ , 则任务 $i$ 完成的时间为 $c_i = \sum_{j=1}^i t_j$ 。任务的平均完成时间(Average Completion Time, ACT)为 $\frac{1}{n} \sum_{i=1}^n c_i$ 。

1) 考虑有四个任务的情况, 每个任务所需时间分别是(4, 2, 8, 1)。若任务的顺序为1, 2, 3, 4, 则ACT是多少?

2) 若任务顺序为2, 1, 4, 3, 则ACT是多少?

3) 创建具有最小ACT的任务序列的贪婪算法分 $n$ 步来构造该任务序列, 在每一步中, 从剩下的任务里选择时间最小的任务。对于1), 利用这种策略获得的任务顺序为4, 2, 1, 3, 这种顺序的ACT是多少?

4) 写一个C++程序实现3)中的贪婪策略, 程序的复杂性应为 $O(n \log n)$ , 试证明之。

5) 证明利用3)中的贪婪算法获得的任务顺序具有最小的ACT。

10. 若有两个工人执行练习9中的 $n$ 个任务, 需将任务分配给他们, 同时他们具有自己的任务执行顺序。任务完成时间及ACT的定义同练习9。使ACT最小化的一种可行的贪婪算法是: 两个工人轮流选择任务, 每次从剩余的任务中选择时间最小的任务。每个人按照自己所选任务的顺序执行任务。对于练习9中的例子, 假定工人1首先选择任务4, 然后工人2选择任务2, 工人1选择任务1, 最后工人2选择任务3。

1) 利用C++程序实现这种策略, 其时间复杂性为多少?

2) 上述的贪婪策略总能获得最小的ACT吗? 证明结论。

11. 1) 考虑有 $m$ 个人可以执行任务, 扩充练习10中的贪婪算法。

2) 算法能保证获得最优解吗? 证明结论。

3) 用C++程序实现此算法, 其复杂性是多少?

12. 考虑例4-4的堆栈折叠问题。

1) 设计一个贪婪算法, 将堆栈折叠为最小数目的子堆栈, 使得每个子堆栈的高度均不超过 $H$ 。

2) 算法总能保证得到数目最少的子堆栈吗? 证明结论。

3) 用C++代码实现1)的算法。

4) 代码的时间复杂性是多少?

13. 编写C++程序实现0/1背包问题, 使用如下启发式方法: 按价值密度非递减的顺序打包。

14. 根据 $k=1$ 的性能受限启发式方法编写一个C++程序来实现0/1背包问题。

15. 对于 $k=1$ 的情况证明用性能受限的启发式方法求解0/1背包问题会发生边界错误。

16. 根据 $k=2$ 的性能受限启发式方法编写一个C++程序来实现0/1背包问题。

17. 考虑  $0 < x_i < 1$  而不是  $x_i \in \{0,1\}$  的连续背包问题。一种可行的贪婪策略是：按价值密度非递减的顺序检查物品，若剩余容量能容下正在考察的物品，将其装入；否则，往背包中装入此物品的一部分。

1) 对于  $n=3$ ,  $w=[100,10,10]$ ,  $p=[20,15,15]$  及  $c=105$ , 上述装入方法获得的结果是什么？

2) 证明这种贪婪算法总能获得最优解。

3) 用一个 C++ 程序实现此算法。

18. 例 13-1 的渴婴问题是练习 17 中连续背包问题的一般化，将练习 17 的贪婪算法用于渴婴问题，算法能保证总能得到最优解吗？证明结论。

19. 1) 证明当且仅当二分图没有覆盖时，图 13-7 的算法找不到覆盖。

2) 给出一个具有覆盖的二分图，使得图 13-7 的算法找不到最小覆盖。

20. 当第一步选择了顶点 1 时，给出图 13-7 的工作过程。

21. 对于二分图覆盖问题设计另外一种贪婪启发式方法，可使用如下贪婪准则：如果  $B$  中的某一个顶点仅被  $A$  中一个顶点覆盖，选择  $A$  中这个顶点；否则，从  $A$  中选择一个顶点，使得它所覆盖的未被覆盖的顶点数目最多。

1) 给出这种贪婪算法的伪代码。

2) 编写一个 C++ 函数作为 Undirected 类的成员来实现上述贪婪算法。

3) 函数的复杂性是多少？

4) 验证代码的正确性。

22. 令  $G$  为无向图， $S$  为  $G$  中顶点的子集，当且仅当  $S$  中的任意两个顶点都有一条边相连时， $S$  为完备子图 (clique)，完备子图的大小即  $S$  中的顶点数目。最大完备子图 (maximum clique) 即具有最大顶点数目的完备子图。在图中寻找最大完备子图的问题 (即最大完备子图问题) 是一个 NP-复杂问题。

1) 给出最大完备子图问题的一种可行的贪婪算法及其伪代码。

2) 给出一个能用 1) 中的启发式算法求解最大完备子图的图例，以及不能用该算法求解的一个图例。

3) 将 1) 中的启发式算法实现为 Undirected::Clique(int C, int m) 共享成员，其中最大完备子图的大小返回到  $m$  中，最大完备子图的顶点返回到  $C$  中。

4) 代码的复杂性是多少？

23. 令  $G$  为一无向图， $S$  为  $G$  中顶点的子集，当且仅当  $S$  中任意两个顶点都无边相连时， $S$  为无关集 (independent set)。最大无关集即是顶点数目最多的无关集。在一幅图中寻找最大无关集是一个 NP-复杂问题。按练习 22 的要求解决最大无关集问题。

24. 对无向图  $G$  着色的方法是：为  $G$  中的顶点编号 ( $\{1,2,\dots\}$ )，使得由一条边相连的两个顶点具有不同的编号。在图的着色问题中，要求利用最少的相互不同的颜色 (编号) 来给图  $G$  着色。图的着色问题也是一个 NP-复杂问题。按练习 22 的要求解决图着色问题。

25. 证明当按路径长度的顺序产生一条最短路径时，所产生的下一条最短路径总是由已产生的一条最短路径扩充一条边得到。

26. 证明对于具有一条或多条具有负长度的边，图 13-11 的贪婪算法不一定能正确地计算出最短路径的长度。

27. 编写一个 Path( $p,s,i$ ) 函数，利用函数 ShortestPaths 计算出的  $p$  值，输出从顶点  $s$  到顶点  $i$  的一条最短路径。函数的复杂性是多少？

28. 若把有向图作为 LinkedwDigraph 类的一个成员，重写程序 13-5，函数应作为该类的一

个成员。函数的复杂性是多少？

29. 若把有向图作为 `LinkedWDigraph` 类的一个成员且仅有  $O(n)$  条边，重写程序 13-5， $L$  用最小堆来实现。函数的复杂性是多少？

30. 从 `Network` 类（见程序 12-15）派生出一个新的模板类 `DNetwork`（有向网络），这个类仅包含应用于有向网络的所有函数。为该类定义一个 `ShortestPaths` 函数，使得它与有向网络的描述形式无关，尤其适用于耗费邻接矩阵及邻接链表描述方法。在函数的实现过程中可利用原来的遍历函数，也可根据需要定义新的遍历函数。函数的复杂性应为  $O(n^2)$ ，其中  $n$  是顶点的数目，试证明之。

\*31. 1) 给出 Prim 算法（如图 13-14 所示）的一种正确性证明。

2) 将图 13-14 细化为一个 C++ 程序 `UNetwork::Prim`，其复杂性应为  $O(n^2)$ 。

3) 证明程序的复杂性确实是  $O(n^2)$ 。

\*32. 1) 证明对于任意连通无向图，Sollin 算法总能找到一个最小耗费生成树。

2) 在 Sollin 算法中，最大的步骤数是多少？试用图中顶点数  $n$  来表示。

3) 编写一个 C++ 程序 `UNetwork::Sollin`，使用 Sollin 算法找到一棵最小生成树。

4) 程序的复杂性是多少？

\*33. 令  $T$  为一棵每条边均带有长度的树（不一定是二叉树）。令  $S$  为  $T$  中顶点的子集，并令  $T/S$  为从  $T$  中删除  $S$  中的顶点所得到的森林。我们希望能找到具有最小走势的子集  $S$ ，使得  $T/S$  中没有从根到叶的距离大于  $d$  的森林。

1) 给出一种寻找最小走势子集  $S$  的贪婪算法（提示：从叶节点开始向根移动）。

2) 证明算法的正确性。

3) 算法的复杂性是多少？如果它不是  $T$  中顶点数的线性函数，则重新设计算法，使其复杂性是线性的。

\*34. 令  $T/S$  表示将  $S$  中的每个顶点复制两份而获得的森林，其中父节点的指针指向一个复本，而另一复本的指针指向其儿子。针对这种情况再做练习 33。

## 13.4 参考及推荐读物

V. Paschos. A Survey of Approximately Optimal Solutions to Some Covering and Packing Problems. *ACM Computing Surveys*, 29, 2, 1997, 171~209。其中描述了几种问题的近似贪婪算法。

B. Moret, H. Shapiro. An Empirical Assessment of Algorithms for Constructing a Minimum Spanning tree. *DIMACS Series in Discrete Mathematics*, 15, 1994, 99~117。它讲述了最小代价生成树问题所使用的贪婪算法的实验估计。





## 第14章 分而治之算法

君主和殖民者们所成功运用的分而治之策略也可以运用到高效率的计算机算法的设计过程中。本章将首先介绍怎样在算法设计领域应用这一古老的策略，然后将利用这一策略解决如下问题：最小最大问题、矩阵乘法、残缺棋盘、排序、选择和—找出二维空间中距离最近的两个点。

本章给出了用来分析分而治之算法复杂性的数学方法，并通过推导最小最大问题和排序问题的复杂性下限来证明分而治之算法对于求解这两种问题是最优的（因为算法的复杂性与下限一致）。

### 14.1 算法思想

分而治之方法与软件设计的模块化方法非常相似。为了解决一个大的问题，可以：1) 把它分成两个或多个更小的问题；2) 分别解决每个小问题；3) 把各小问题的解答组合起来，即可得到原问题的解答。小问题通常与原问题相似，可以递归地使用分而治之策略来解决。

例14-1 [找出伪币] 给你一个装有16个硬币的袋子。16个硬币中有一个是伪造的，并且那个伪造的硬币比真的硬币要轻一些。你的任务是找出这个伪造的硬币。为了帮助你完成这一任务，将提供一台可用来比较两组硬币重量的仪器，利用这台仪器，可以知道两组硬币的重量是否相同。

比较硬币1与硬币2的重量。假如硬币1比硬币2轻，则硬币1是伪造的；假如硬币2比硬币1轻，则硬币2是伪造的。这样就完成了任务。假如两硬币重量相等，则比较硬币3和硬币4。同样，假如有一个硬币轻一些，则寻找伪币的任务完成。假如两硬币重量相等，则继续比较硬币5和硬币6。按照这种方式，可以最多通过8次比较来判断伪币的存在并找出这一伪币。

另外一种方法就是利用分而治之方法。假如把16硬币的例子看成一个大的问题。第一步，把这一问题分成两个小问题。随机选择8个硬币作为第一组称为A组，剩下的8个硬币作为第二组称为B组。这样，就把16个硬币的问题分成两个8硬币的问题来解决。第二步，判断A和B组中是否有伪币。可以利用仪器来比较A组硬币和B组硬币的重量。假如两组硬币重量相等，则可以判断伪币不存在。假如两组硬币重量不相等，则存在伪币，并且可以判断它位于较轻的那一组硬币中。最后，在第三步中，用第二步的结果得出原先16个硬币问题的答案。若仅仅判断硬币是否存在，则第三步非常简单。无论A组还是B组中有伪币，都可以推断这16个硬币中存在伪币。因此，仅仅通过一次重量的比较，就可以判断伪币是否存在。

现在假设需要识别出这一伪币。把两个或三个硬币的情况作为不可再分的小问题。注意如果只有一个硬币，那么不能判断出它是否就是伪币。在一个小问题中，通过将一个硬币分别与其他两个硬币比较，最多比较两次就可以找到伪币。

这样，16硬币的问题就被分为两个8硬币（A组和B组）的问题。通过比较这两组硬币的重量，可以判断伪币是否存在。如果没有伪币，则算法终止。否则，继续划分这两组硬币来寻找伪币。假设B是轻的那一组，因此再把它分成两组，每组有4个硬币。称其中一组为B1，另一组为B2。比较这两组，肯定有一组轻一些。如果B1轻，则伪币在B1中，再将B1又分成两组，

每组有两个硬币，称其中一组为  $B1a$ ，另一组为  $B1b$ 。比较这两组，可以得到一个较轻的组。由于这个组只有两个硬币，因此不必再细分。比较组中两个硬币的重量，可以立即知道哪一个硬币轻一些。较轻的硬币就是所要找的伪币。

例14-2 [金块问题] 有一个老板有一袋金块。每个月将有两名雇员会因其优异的表现分别被奖励一个金块。按规矩，排名第一的雇员将得到袋中最重的金块，排名第二的雇员将得到袋中最轻的金块。根据这种方式，除非有新的金块加入袋中，否则第一名雇员所得到的金块总是比第二名雇员所得到的金块重。如果有新的金块周期性的加入袋中，则每个月都必须找出最轻和最重的金块。假设有一台比较重量的仪器，我们希望用最少的比较次数找出最轻和最重的金块。

假设袋中有  $n$  个金块。可以用函数 Max（程序1-31）通过  $n-1$  次比较找到最重的金块。找到最重的金块后，可以从余下的  $n-1$  个金块中用类似的方法通过  $n-2$  次比较找出最轻的金块。这样，比较的总次数为  $2n-3$ 。程序2-26和2-27是另外两种方法，前者需要进行  $2n-2$  次比较，后者最多需要进行  $2n-2$  次比较。

下面用分而治之方法对这个问题进行求解。当  $n$  很小时，比如说， $n=2$ ，识别出最重和最轻的金块，一次比较就足够了。当  $n$  较大时（ $n>2$ ），第一步，把这袋金块平分成两个小袋  $A$  和  $B$ 。第二步，分别找出在  $A$  和  $B$  中最重和最轻的金块。设  $A$  中最重和最轻的金块分别为  $H_A$  与  $L_A$ ，以此类推， $B$  中最重和最轻的金块分别为  $H_B$  和  $L_B$ 。第三步，通过比较  $H_A$  和  $H_B$ ，可以找到所有金块中最重的；通过比较  $L_A$  和  $L_B$ ，可以找到所有金块中最轻的。在第二步中，若  $n>2$ ，则递归地应用分而治之方法。

假设  $n=8$ 。这个袋子被平分为各有4个金块的两个袋子  $A$  和  $B$ 。为了在  $A$  中找出最重和最轻的金块， $A$  中的4个金块被分成两组  $A1$  和  $A2$ 。每一组有两个金块，可以用一次比较在  $A$  中找出较重的金块  $H_{A1}$  和较轻的金块  $L_{A1}$ 。经过另外一次比较，又能找出  $H_{A2}$  和  $L_{A2}$ 。现在通过比较  $H_{A1}$  和  $H_{A2}$ ，能找出  $H_A$ ；通过  $L_{A1}$  和  $L_{A2}$  的比较找出  $L_A$ 。这样，通过4次比较可以找到  $H_A$  和  $L_A$ 。同样需要另外4次比较来确定  $H_B$  和  $L_B$ 。通过比较  $H_A$  和  $H_B$ （ $L_A$  和  $L_B$ ），就能找出所有金块中最重和最轻的。因此，当  $n=8$  时，这种分而治之的方法需要10次比较。如果使用程序1-31，则需要13次比较。如果使用程序2-26和2-27，则最多需要14次比较。

设  $c(n)$  为使用分而治之方法所需要的比较次数。为了简便，假设  $n$  是2的幂。当  $n=2$  时， $c(n)=1$ 。对于较大的  $n$ ， $c(n)=2c(n/2)+2$ 。当  $n$  是2的幂时，使用迭代方法（见例2-20）可知  $c(n)=3n/2-2$ 。在本例中，使用分而治之方法比逐个比较的方法少用了25%的比较次数。

例14-3 [矩阵乘法] 两个  $n \times n$  阶的矩阵  $A$  与  $B$  的乘积是另一个  $n \times n$  阶矩阵  $C$ ， $C$  可表示为

$$C(i,j) = \sum_{k=1}^n A(i,k) * B(k,j), \quad 1 \leq i \leq n, \quad 1 \leq j \leq n \quad (14-1)$$

假如每一个  $C(i,j)$  都用此公式计算，则计算  $C$  所需要的操作次数为  $n^3 m + n^2 (n-1)a$ ，其中  $m$  表示一次乘法， $a$  表示一次加法或减法。

为了得到两个矩阵相乘的分而治之算法，需要：1) 定义一个小问题，并指明小问题是如何进行乘法运算的；2) 确定如何把一个大的问题划分成较小的问题，并指明如何对这些较小的问题进行乘法运算；3) 最后指出如何根据小问题的结果得到大问题的结果。为了使讨论简便，假设  $n$  是2的幂（也就是说， $n$  是1, 2, 4, 8, 16, ...）。

首先，假设  $n=1$  时是一个小问题， $n>1$  时为一个大问题。后面将根据需要随时修改这个假设。对于  $1 \times 1$  阶的小矩阵，可以通过将两矩阵中的两个元素直接相乘而得到结果。

考察一个  $n>1$  的大问题。可以将这样的矩阵分成4个  $n/2 \times n/2$  阶的矩阵  $A_1, A_2, A_3$ ，和  $A_4$ ，如

图14-1a 所示。当 $n$  大于1且 $n$  是2的幂时， $n/2$ 也是2的幂。因此较小矩阵也满足前面对矩阵大小的假设。矩阵 $B_i$ 和 $C_i$ 的定义与此类似， $1 \leq i \leq 4$ 。矩阵乘积的结果见图14-1b。

$$\begin{array}{c}
 \begin{array}{cc}
 n/2 & n/2 \\
 \begin{array}{|c|c|}
 \hline
 A_1 & A_2 \\
 \hline
 A_3 & A_4 \\
 \hline
 \end{array}
 &
 \begin{array}{|c|c|}
 \hline
 B_1 & B_2 \\
 \hline
 B_3 & B_4 \\
 \hline
 \end{array}
 \\
 n/2 & n/2
 \end{array}
 \end{array}
 \quad
 \begin{array}{c}
 \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} * \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} C_1 & C_2 \\ C_3 & C_4 \end{bmatrix}
 \end{array}
 \end{array}
 \quad
 \begin{array}{c}
 \text{a)} \\
 \text{b)}
 \end{array}$$

图14-1 把一个矩阵划分成几个小矩阵

a) 把A分为4个小矩阵 b)  $A*B = C$

可以利用公式 (14-1) 来证明以下公式：

$$C_1 = A_1B_1 + A_2B_3 \quad (14-2)$$

$$C_2 = A_1B_2 + A_2B_4 \quad (14-3)$$

$$C_3 = A_3B_1 + A_4B_3 \quad (14-4)$$

$$C_4 = A_3B_2 + A_4B_4 \quad (14-5)$$

根据上述公式，经过8次 $n/2 \times n/2$ 阶矩阵乘法 and 4次 $n/2 \times n/2$ 阶矩阵的加法，就可以计算出 $A$ 与 $B$ 的乘积。因此，这些公式能帮助我们实现分而治之算法。在算法的第二步，将递归使用分而治之算法把8个小矩阵再细分（见程序2-19）。算法的复杂性为 $\Theta(n^3)$ ，此复杂性与程序2-24直接使用公式 (14-1) 所得到的复杂性是一样的。事实上，由于矩阵分割和再组合所花费的额外开销，使用分而治之算法得出结果的时间将比用程序2-24还要长。

为了得到更快的算法，需要简化矩阵分割和再组合这两个步骤。一种方案是使用 Strassen 方法得到7个小矩阵。这7个小矩阵为矩阵 $D, E, \dots, J$ ，它们分别定义为：

$$D = A_1(B_2 - B_4)$$

$$E = A_4(B_3 - B_1)$$

$$F = (A_3 + A_4)B_1$$

$$G = (A_1 + A_2)B_4$$

$$H = (A_3 - A_1)(B_1 + B_2)$$

$$I = (A_2 - A_4)(B_3 + B_4)$$

$$J = (A_1 + A_4)(B_1 + B_4)$$

矩阵 $D$ 到 $J$ 可以通过7次矩阵乘法，6次矩阵加法，和4次矩阵减法计算得出。前述的4个小矩阵可以由矩阵 $D$ 到 $J$ 通过6次矩阵加法和两次矩阵减法得出，方法如下：

$$C_1 = E + I + J - G$$

$$C_2 = D + G$$

$$C_3 = E + F$$

$$C_4 = D + H + J - F$$

用上述方案来解决 $n=2$ 的矩阵乘法。将某矩阵 $A$ 和 $B$ 相乘得结果 $C$ ，如下所示：

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

因为 $n>1$ ，所以将A、B两矩阵分别划分为4个小矩阵，如图14-1a所示。每个矩阵为 $1 \times 1$ 阶，仅包含一个元素。 $1 \times 1$ 阶矩阵的乘法为小问题，因此可以直接进行运算。利用计算 $D \sim J$ 的公式，得：

$$D=1(6-8)=-2$$

$$E=4(7-5)=8$$

$$F=(3+4)5=35$$

$$G=(1+2)8=24$$

$$H=(3-1)(5+6)=22$$

$$I=(2-4)(7+8)=-30$$

$$J=(1+4)(5+8)=65$$

根据以上结果可得：

$$C_1 = 8 - 30 + 65 - 24 = 19$$

$$C_2 = -2 + 24 = 22$$

$$C_3 = 8 + 35 = 43$$

$$C_4 = -2 + 22 + 65 - 35 = 50$$

对于上面这个 $2 \times 2$ 的例子，使用分而治之算法需要7次乘法和18次加/减法运算。而直接使用公式(14-1)，则需要8次乘法和7次加/减法。要想使分而治之算法更快一些，则一次乘法所花费的时间必须比11次加/减法的时间要长。

假定Strassen矩阵分割方案仅用于 $n \geq 8$ 的矩阵乘法，而对于 $n < 8$ 的矩阵乘法则直接利用公式(14-1)进行计算。则 $n=8$ 时， $8 \times 8$ 矩阵相乘需要7次 $4 \times 4$ 矩阵乘法和18次 $4 \times 4$ 矩阵加/减法。每次矩阵乘法需花费 $64m+48a$ 次操作，每次矩阵加法或减法需花费 $16a$ 次操作。因此总的操作次数为 $7(64m+48a)+18(16a)=448m+624a$ 。而使用直接计算方法，则需要 $512m+448a$ 次操作。要使Strassen方法比直接计算方法快，至少要求 $512-448$ 次乘法的开销比 $624-448$ 次加/减法的开销大。或者说一次乘法的开销应该大于近似2.75次加/减法的开销。

假定 $n < 16$ 的矩阵是一个“小”问题，Strassen的分解方案仅仅用于 $n \geq 16$ 的情况，对于 $n < 16$ 的矩阵相乘，直接利用公式(14-1)。则当 $n=16$ 时使用分而治之算法需要 $7(512m+448a)+18(64a)=3584m+4288a$ 次操作。直接计算时需要 $4096m+3840a$ 次操作。若一次乘法的开销与一次加/减法的开销相同，则Strassen方法需要7872次操作及用于问题分解的额外时间，而直接计算方法则需要7936次操作加上程序中执行for循环以及其他语句所花费的时间。即使直接计算方法所需要的操作次数比Strassen方法少，但由于直接计算方法需要更多的额外开销，因此它也不见得会比Strassen方法快。

$n$ 的值越大，Strassen方法与直接计算方法所用的操作次数的差异就越大，因此对于足够大的 $n$ ，Strassen方法将更快。设 $t(n)$ 表示使用Strassen分而治之方法所需的时间。因为大的矩阵会被递归地分割成小矩阵直到每个矩阵的大小小于或等于 $k$ （ $k$ 至少为8，也许更大，具体值由计算机的性能决定）， $t$ 的递归表达式如下：

$$t(n) = \begin{cases} d & n \leq k \\ 7t(n/2) + cn^2 & n > k \end{cases} \quad (14-6)$$

这里 $cn^2$ 表示完成18次 $n/2 \times n/2$ 阶矩阵加/减法以及把大小为 $n$ 的矩阵分割成小矩阵所需要

的时间。用迭代方法计算,可得 $t(n)=\Theta(n\log_2^7)$ 。因为 $\log_2^7 \approx 2.81$ ,所以与直接计算方法的复杂度 $\Theta(n^3)$ 相比,分而治之矩阵乘法算法有较大的改进。

#### 注意事项

分而治之方法很自然地导致了递归算法的使用。在许多例子里,这些递归算法在递归程序中得到了很好的运用。实际上,在许多情况下,所有为了得到一个非递归程序的企图都会导致采用一个模拟递归栈。不过在有些情况下,不使用这样的递归栈而采用一个非递归程序来完成分而治之算法也是可能的,并且在这种方式下,程序得到结果的速度会比递归方式更快。解决金块问题的分而治之算法(例14-2)和归并排序方法(14.3节)就可以不利用递归而通过一个非递归程序来更快地完成。

**例14-4 [金块问题]**用例14-2的算法寻找8个金块中最轻和最重金块的工作可以用图14-2中的二叉树来表示。这棵树的叶子分别表示8个金块( $a, b, \dots, h$ ),每个阴影节点表示一个包含其子树中所有叶子的问题。因此,根节点A表示寻找8个金块中最轻、最重金块的问题,而节点B表示找出 $a, b, c$ 和 $d$ 这4个金块中最轻和最重金块的问题。算法从根节点开始。由根节点表示的8金块问题被划分成由节点B和C所表示的两个4金块问题。在B节点,4金块问题被划分成由D和E所表示的2金块问题。可通过比较金块 $a$ 和 $b$ 哪一个较重来解决D节点所表示的2金块问题。在解决了D和E所表示的问题之后,可以通过比较D和E中所找到的轻金块和重金块来解决B表示的问题。接着在F, G和C上重复这一过程,最后解决问题A。

可以将递归的分而治之算法划分成以下的步骤:

- 1) 从图14-2中的二叉树由根至叶的过程中把一个大问题划分成许多个小问题,小问题的大小为1或2。
- 2) 比较每个大小为2的问题中的金块,确定哪一个较重和哪一个较轻。在节点D、E、F和G上完成这种比较。大小为1的问题中只有一个金块,它既是最轻的金块也是最重的金块。
- 3) 对较轻的金块进行比较以确定哪一个金块最轻,对较重的金块进行比较以确定哪一个金块最重。对于节点A到C执行这种比较。

根据上述步骤,可以得出程序14-1的非递归代码。该程序用于寻找到数组 $w[0:n-1]$ 中的最小数和最大数,若 $n < 1$ ,则程序返回false,否则返回true。

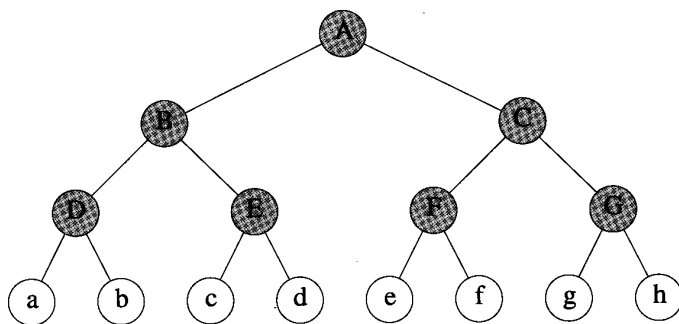


图14-2 找出8个金块中最轻和最重的金块

当 $n = 1$ 时,程序14-1给Min和Max置初值以使 $w[\text{Min}]$ 是最小的重量, $w[\text{Max}]$ 为最大的重量。首先处理 $n = 1$ 的情况。若 $n > 1$ 且为奇数,第一个重量 $w[0]$ 将成为最小值和最大值的候选值,因此将有偶数个重量值 $w[1:n-1]$ 参与for循环。当 $n$ 是偶数时,首先将两个重量值放在for循环外进行

比较，较小和较大的重量值分别置为Min和Max，因此也有偶数个重量值 $w[2:n-1]$ 参与for循环。

在for 循环中，外层if 通过比较确定 $(w[i], w[i+1])$ 中的较大和较小者。此工作与前面提到的分而治之算法步骤中的2) 相对应，而内层的if负责找出较小重量值和较大重量值中的最小值和最大值，这个工作对应于3)。

for 循环将每一对重量值中较小值和较大值分别与当前的最小值  $w[\text{Min}]$ 和最大值 $w[\text{Max}]$ 进行比较，根据比较结果来修改Min和Max（如果必要）。

下面进行复杂性分析。注意到当n为偶数时，在for 循环外部将执行一次比较而在for循环内部执行 $3(n/2-1)$ 次比较，比较的总次数为 $3n/2-2$ 。当n 为奇数时，for循环外部没有执行比较，而内部执行了 $3(n-1)/2$ 次比较。因此无论n 为奇数或偶数，当 $n>0$ 时，比较的总次数为 $\lceil 3n/2 \rceil - 2$ 次。

程序14-1 找出最小值和最大值的非递归程序

```
template<class T>
bool MinMax(T w[], int n, T& Min, T& Max)
{// 寻找w[0:n-1]中的最小和最大值
// 如果少于一个元素，则返回 false
// 特殊情形：n <= 1
if (n < 1) return false;
if (n == 1) {Min = Max = 0;
return true;}

//对Min 和Max进行初始化
int s; // 循环起点
if (n % 2) { // n 为奇数
Min = Max = 0;
s = 1;}
else { // n为偶数，比较第一对
if (w[0] > w[1]) {
Min = 1;
Max = 0;}
else {Min = 0;
Max = 1;}
s = 2;}

// 比较余下的数对
for (int i = s; i < n; i += 2) {
// 寻找w[i] 和w[i+1]中的较大者
// 然后将较大者与 w[Max]进行比较
// 将较小者与 w[Min]进行比较
if (w[i] > w[i+1]) {
if (w[i] > w[Max]) Max = i;
if (w[i+1] < w[Min]) Min = i + 1;}
else {
if (w[i+1] > w[Max]) Max = i + 1;
if (w[i] < w[Min]) Min = i;}
}

return true;
}
```



## 练习

1. 将例14-1的分而治之算法扩充到 $n > 1$ 个硬币的情形。需要进行多少次重量的比较？
2. 考虑例14-1的伪币问题。假设把条件“伪币比真币轻”改为“伪币与真币的重量不同”，同样假定袋中有 $n$ 个硬币。
  - 1) 给出相应分而治之算法的形式化描述，该算法可输出信息“不存在伪币”或找出伪币。算法应递归地将大的问题划分成两个较小的问题。需要多少次比较才能找到伪币（如果存在伪币）？
  - 2) 重复1)，但把大问题划分为三个较小问题。
3. 1) 编写一个C++ 程序，实现例14-2中寻找 $n$ 个元素中最大值和最小值的两种方案。使用递归来完成分而治之方案。
  - 2) 程序2-26和2-27是另外两个寻找 $n$ 个元素中最大值和最小值的代码。试分别计算出每段程序所需要的最少和最大比较次数。
  - 3) 在 $n$ 分别等于100，1000或10 000的情况下，比较1) 2) 中的程序和程序14-1的运行时间。对于程序2-27，使用平均时间和最坏情况下的时间。1) 中的程序和程序2-26应具有相同的平均时间和最坏情况下的时间。
  - 4) 注意到如果比较操作的开销不是很高，分而治之算法在最坏情况下不会比其他算法优越，为什么？它的平均时间优于程序2-27吗？为什么？
4. 证明直接运用公式(14-2) ~ (14-5) 得出结果的矩阵乘法的分而治之算法的复杂性为 $\Theta(n^3)$ 。因此相应的分而治之程序将比程序2-24要慢。
5. 用迭代的方法来证明公式(14-6)的递归值为 $\Theta(n^{\log_2 7})$ 。
- \*6. 编写Strassen矩阵乘法程序。利用不同的 $k$ 值（见公式(14-6)）进行实验，以确定 $k$ 为何值时程序性能最佳。比较程序及程序2-24的运行时间。可取 $n$ 为2的幂来进行比较。
7. 当 $n$ 不是2的幂时，可以通过增加矩阵的行和列来得到一个大小为2的幂的矩阵。假设使用最少的行数和列数将矩阵扩充为 $m$ 阶矩阵，其中 $m$ 为2的幂。
  - 1) 求 $m/n$ 。
  - 2) 可使用哪些矩阵项组成新的行和列，以使新矩阵 $A'$ 和 $B'$ 相乘时，原来的矩阵 $A$ 和 $B$ 相乘的结果会出现在 $C'$ 的左上角？
  - 3) 使用Strassen方法计算 $A' * B'$ 所需要的时间为 $\Theta(m^{2.81})$ 。给出以 $n$ 为变量的运行时间表达式。

## 14.2 应用

## 14.2.1 残缺棋盘

残缺棋盘 (defective chessboard) 是一个有 $2^k \times 2^k$ 个方格的棋盘，其中恰有一个方格残缺。图14-3给出 $k=2$ 时各种可能的残缺棋盘，其中残缺的方格用阴影表示。注意当 $k=0$ 时，仅存在一种可能的残缺棋盘（如图14-3a所示）。事实上，对于任意 $k$ ，恰好存在 $2^k$ 种不同的残缺棋盘。

残缺棋盘的问题要求用三格板 (triominoes) 覆盖残缺棋盘（如图14-4所示）。在此覆盖中，两个三格板不能重叠，三格板不能覆盖残缺方格，但必须覆盖其他所有的方格。在这种限制条件下，所需要的三格板总数为 $(2^{2k} - 1)/3$ 。可以验证 $(2^{2k} - 1)/3$ 是一个整数。 $k$ 为0的残缺棋盘很容易被覆盖，因为它没有非残缺的方格，用于覆盖的三格板的数目为0。当 $k=1$ 时，正好存在3个



非残缺的方格，并且这三个方格可用图 14-4 中的某一方向的三格板来覆盖。

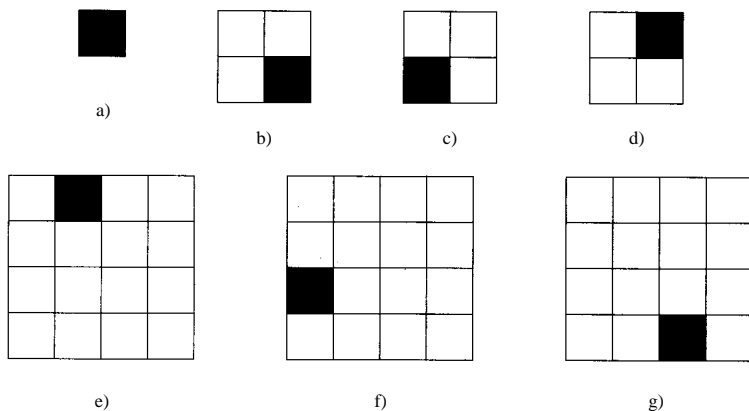


图14-3 残缺棋盘

a)  $k=0$  b)  $k=1$  c)  $k=1$  d)  $k=1$  e)  $k=2$  f)  $k=2$  g)  $k=2$

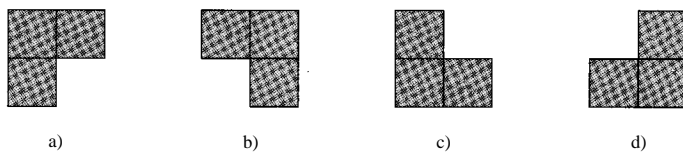


图14-4 不同方向的三格板

用分而治之的方法可以很好地解决残缺棋盘问题。这一方法可将覆盖  $2^k \times 2^k$  残缺棋盘的问题转化为覆盖较小残缺棋盘的问题。 $2^k \times 2^k$  棋盘一个很自然的划分方法就是将它划分为如图 14-5a 所示的 4 个  $2^{k-1} \times 2^{k-1}$  棋盘。注意到当完成这种划分后，4 个小棋盘中仅仅有一个棋盘存在残缺方格（因为原来的  $2^k \times 2^k$  棋盘仅仅有一个残缺方格）。首先覆盖其中包含残缺方格的  $2^{k-1} \times 2^{k-1}$  残缺棋盘，然后把剩下的 3 个小棋盘转变为残缺棋盘，为此将一个三格板放在由这 3 个小棋盘形成的角上，如图 14-5b 所示，其中原  $2^k \times 2^k$  棋盘中的残缺方格落入左上角的  $2^{k-1} \times 2^{k-1}$  棋盘。可以

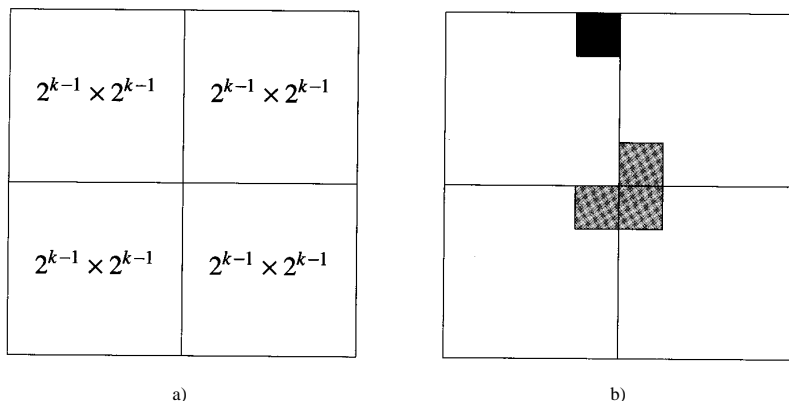


图14-5 分割  $2^k \times 2^k$  棋盘

a) 划分方法 b) 放置三格板

采用这种分割技术递归地覆盖  $2^k \times 2^k$  残缺棋盘。当棋盘的大小减为  $1 \times 1$  时，递归过程终止。此时  $1 \times 1$  的棋盘中仅仅包含一个方格且此方格残缺，所以无需放置三格板。

可以将上述分而治之的算法编写成一个递归的 C++ 函数 TileBoard (见程序 14-2)。该函数定义了一个全局的二维整数数组变量 Board 来表示棋盘。Board[0][0] 表示棋盘中左上角的方格。该函数还定义了一个全局整数变量 tile，其初始值为 0。函数的输入参数如下：

- tr 棋盘中左上角方格所在行。
- tc 棋盘中左上角方格所在列。
- dr 残缺方块所在行。
- dl 残缺方块所在列。
- size 棋盘的行数或列数。

TileBoard 函数的调用格式为 TileBoard ( 0,0, dr, dc,size)，其中  $\text{size}=2^k$ 。覆盖残缺棋盘所需要的三格板数目为  $(\text{size}^2 - 1)/3$ 。函数 TileBoard 用整数 1 到  $(\text{size}^2 - 1)/3$  来表示这些三格板，并用三格板的标号来标记被该三格板覆盖的非残缺方格。

令  $t(k)$  为函数 TileBoard 覆盖一个  $2^k \times 2^k$  残缺棋盘所需要的时间。当  $k=0$  时，size 等于 1，覆盖它将花费常数时间  $d$ 。当  $k > 0$  时，将进行 4 次递归的函数调用，这些调用需花费的时间为  $4t(k-1)$ 。除了这些时间外，if 条件测试和覆盖 3 个非残缺方格也需要时间，假设用常数  $c$  表示这些额外时间。可以得到以下递归表达式：

$$t(k) = \begin{cases} d & k=0 \\ 4t(k-1) + c & k>0 \end{cases} \quad (14-7)$$

程序 14-2 覆盖残缺棋盘

```
void TileBoard(int tr, int tc, int dr, int dc, int size)
{
    // 覆盖残缺棋盘
    if (size == 1) return;
    int t = tile++; // 所使用的三格板的数目
    s = size/2; // 象限大小

    //覆盖左上象限
    if (dr < tr + s && dc < tc + s)
        // 残缺方格位于本象限
        TileBoard(tr, tc, dr, dc, s);
    else { // 本象限中没有残缺方格
        // 把三格板 t 放在右下角
        Board[tr + s - 1][tc + s - 1] = t;
        // 覆盖其余部分
        TileBoard(tr, tc, tr+s-1, tc+s-1, s);
    }

    //覆盖右上象限
    if (dr < tr + s && dc >= tc + s)
        // 残缺方格位于本象限
        TileBoard(tr, tc+s, dr, dc, s);
    else { // 本象限中没有残缺方格
        // 把三格板 t 放在左下角
        Board[tr + s - 1][tc + s] = t;
        // 覆盖其余部分
    }
}
```

```

    TileBoard(tr, tc+s, tr+s-1, tc+s, s);}

//覆盖左下象限
if (dr >= tr + s && dc < tc + s)
    // 残缺方格位于本象限
    TileBoard(tr+s, tc, dr, dc, s);
else { // 把三格板 t 放在右上角
    Board[tr + s][tc + s - 1] = t;
    // 覆盖其余部分
    TileBoard(tr+s, tc, tr+s, tc+s-1, s);}

// 覆盖右下象限
if (dr >= tr + s && dc >= tc + s)
    // 残缺方格位于本象限
    TileBoard(tr+s, tc+s, dr, dc, s);
else { // 把三格板 t 放在左上角
    Board[tr + s][tc + s] = t;
    // 覆盖其余部分
    TileBoard(tr+s, tc+s, tr+s, tc+s, s);}
}

void OutputBoard(int size)
{
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++)
            cout << setw(5) << Board[i][j];
        cout << endl;
    }
}

```

可以用迭代的方法来计算这个表达式（见例 2-20），可得  $t(k) = \Theta(4^k) = \Theta$ （所需的三格板的数目）。由于必须花费至少  $\Theta(1)$  的时间来放置每一块三格表，因此不可能得到一个比分而治之算法更快的算法。

### 14.2.2 归并排序

可以运用分而治之的方法来解决排序问题，该问题是将  $n$  个元素排成非递减顺序。分而治之的方法通常用以下的步骤来进行排序算法：若  $n$  为 1，算法终止；否则，将这一元素集合分割成两个或更多个子集合，对每一个子集合分别排序，然后将排好序的子集合归并为一个集合。

假设仅将  $n$  个元素的集合分成两个子集合。现在需要确定如何进行子集合的划分。一种可能性就是把前面  $n-1$  个元素放到第一个子集中（称为  $A$ ），最后一个元素放到第二个子集里（称为  $B$ ）。按照这种方式对  $A$  递归地进行排序。由于  $B$  仅含一个元素，所以它已经排序完毕，在  $A$  排完序后，只需要用程序 2-10 中的函数 insert 将  $A$  和  $B$  合并起来。把这种排序算法与 InsertionSort（见程序 2-15）进行比较，可以发现这种排序算法实际上就是插入排序的递归算法。该算法的复杂性为  $O(n^2)$ 。

把  $n$  个元素划分成两个子集合的另一种方法是将含有最大值的元素放入  $B$ ，剩下的放入  $A$  中。然后  $A$  被递归排序。为了合并排序后的  $A$  和  $B$ ，只需要将  $B$  添加到  $A$  中即可。假如用函数 Max（见程序 1-31）来找出最大元素，这种排序算法实际上就是 SelectionSort（见程序 2-7）的递归算法。

假如用冒泡过程（见程序 2-8）来寻找最大元素并把它移到最右边的位置，这种排序算法就是 BubbleSort（见程序 2-9）的递归算法。这两种递归排序算法的复杂性均为  $\Theta(n^2)$ 。若一旦发现  $A$  已经被排好序就终止对  $A$  进行递归分割，则算法的复杂性为  $O(n^2)$ （见例 2-16 和 2-17）。

上述分割方案将  $n$  个元素分成两个极不平衡的集合  $A$  和  $B$ 。 $A$  有  $n-1$  个元素，而  $B$  仅含一个元素。下面来看一看采用平衡分割法会发生什么情况： $A$  集合中含有  $n/k$  个元素， $B$  中包含其余的元素。递归地使用分而治之的方法对  $A$  和  $B$  进行排序。然后采用一个被称之为归并（merge）的过程，将已排好序的  $A$  和  $B$  合并成一个集合。

例 14-5 考虑 8 个元素，值分别为  $[10, 4, 6, 3, 8, 2, 5, 7]$ 。如果选定  $k=2$ ，则  $[10, 4, 6, 3]$  和  $[8, 2, 5, 7]$  将被分别独立地排序。结果分别为  $[3, 4, 6, 10]$  和  $[2, 5, 7, 8]$ 。从两个序列的头部开始归并这两个已排序的序列。元素 2 比 3 更小，被移到结果序列；3 与 5 进行比较，3 被移入结果序列；4 与 5 比较，4 被放入结果序列；5 和 6 比较，...

如果选择  $k=4$ ，则序列  $[10, 4]$  和  $[6, 3, 8, 2, 5, 7]$  将被排序。排序结果分别为  $[4, 10]$  和  $[2, 3, 5, 6, 7, 8]$ 。当这两个排好序的序列被归并后，即可得所需要的排序序列。

图 14-6 给出了分而治之的排序算法的伪代码。算法中子集合的数目为 2， $A$  中含有  $n/k$  个元素。

```
template<class T>
void sort(T E, int n)
{//对E中的n个元素进行排序，k为全局变量
    if (n >= k) {
        i = n/k;
        j = n-i;
        令A 包含E中的前 i 个元素
        令 B 包含E中余下的 j 个元素
        sort(A,i);
        sort(B,j);
        merge(A,B,E,i,j); //把A 和 B 合并到 E
    }
    else 使用插入排序算法对 E 进行排序
}
```

图 14-6 分而治之的排序算法的伪代码

从对归并过程的简略描述中，可以明显地看出归并  $n$  个元素所需要的时间为  $O(n)$ 。设  $t(n)$  为分而治之的排序算法（如图 14-6 所示）在最坏情况下所需花费的时间，则有以下递推公式：

$$t(n) = \begin{cases} d & n < k \\ t(n/k) + t(n - n/k) + cn & n \geq k \end{cases}$$

其中  $c$  和  $d$  为常数。当  $n/k \sim n - n/k$  时， $t(n)$  的值最小。因此当  $k=2$  时，也就是说，当两个子集合所包含的元素个数近似相等时， $t(n)$  最小，即当所划分的子集合大小接近时，分而治之的算法通常具有最佳性能。

在  $t(n)$  的递推公式中，取  $k=2$ ，可得到如下递推公式：

$$t(n) = \begin{cases} d & n \leq 1 \\ t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + cn & n > 1 \end{cases}$$

由于上面递推公式中存在两个取整运算，因此计算该递推公式比较困难。如果仅考虑  $n$  为 2 的幂，递推公式可简化为：

$$t(n) = \begin{cases} d & n \leq 1 \\ 2t(n/2) + cn & n > 1 \end{cases}$$

可以用迭代方法来计算这一递推方式，结果为  $t(n) = \Theta(n \log n)$ 。虽然这个结果是在  $n$  为 2 的幂时得到的，但对于所有的  $n$ ，这一结果也是有效的，因为  $t(n)$  是  $n$  的非递减函数。 $t(n) = \Theta(n \log n)$  给出了归并排序的最好和最坏情况下的复杂性。由于最好和最坏情况下的复杂性是一样的，因此归并排序的平均复杂性为  $t(n) = \Theta(n \log n)$ 。

#### 1. 将图 14-6 改写为 C++ 代码

图 14-6 中  $k=2$  的排序方法被称为归并排序 (merge sort)，或更精确地说是二路归并排序 (two-way merge sort)。下面根据图 14-6 中  $k=2$  的情况 (归并排序) 来编写对  $n$  个元素进行排序的 C++ 函数。一种最简单的方法就是将元素存储在链表中 (即作为类 chain 的成员 (程序 3-8))。在这种情况下，通过移到第  $n/2$  个节点并打断此链，可将  $E$  分成两个大致相等的链表。归并过程应能将两个已排序的链表归并在一起。如果希望把所得到的 C++ 程序与堆排序和插入排序进行性能比较，那么就不能使用链表来实现归并排序，因为后两种排序方法中都没有使用链表。

为了能与前面讨论过的排序函数作比较，归并排序函数必须用一个数组  $a$  来存储元素集合  $E$ ，并在  $a$  中返回排序后的元素序列。为此按照下述过程来对图 14-6 的伪代码进行细化：当集合  $E$  被化分成两个子集合时，可以不必把两个子集合的元素分别复制到  $A$  和  $B$  中，只需简单地在集合  $E$  中保持两个子集合的左右边界即可。接下来对  $a$  中的初始序列进行排序，并将所得到的排序序列归并到一个新数组  $b$  中，最后将它们复制到  $a$  中。图 14-6 的改进版见图 14-7。

```
template<class T>
MergeSort( T a[], int left, int right)
{ //对a[left:right]中的元素进行排序
    if (left < right) { //至少两个元素
        int i = (left + right)/2; //中心位置
        MergeSort(a, left, i);
        MergeSort(a, i+1, right);
        Merge(a, b, left, i, right); //从a 合并到 b
        Copy(b, a, left, right); //结果放回 a
    }
}
```

图 14-7 分而治之排序算法的改进

可以从很多方面来改进图 14-7 的性能，例如，可以容易地消除递归。如果仔细地检查图 14-7 中的程序，就会发现其中的递归只是简单地重复分割元素序列，直到序列的长度变成 1 为止。当序列的长度变为 1 时即可进行归并操作，这个过程可以用  $n$  为 2 的幂来很好地描述。长度为 1 的序列被归并为长度为 2 的有序序列；长度为 2 的序列接着被归并为长度为 4 的有序序列；这个过程不断地重复直到归并为长度为  $n$  的序列。图 14-8 给出  $n=8$  时的归并 (和复制) 过程，方括号表示一个已排序序列的首和尾。

初始序列	[8] [4] [5] [6] [2] [1] [7] [3]
归并到 b	[4 8] [5 6] [1 2] [3 7]
复制到 a	[4 8] [5 6] [1 2] [3 7]
归并到 b	[4 5 6 8] [1 2 3 7]
复制到 a	[4 5 6 8] [1 2 3 7]
归并到 b	[1 2 3 4 5 6 7 8]
复制到 a	[1 2 3 4 5 6 7 8]

图14-8 归并排序的例子

另一种二路归并排序算法是这样的：首先将每两个相邻的大小为 1 的子序列归并，然后对上一次归并所得到的大小为 2 的子序列进行相邻归并，如此反复，直至最后归并到一个序列，归并过程完成。通过轮流地将元素从 a 归并到 b 并从 b 归并到 a，可以虚拟地消除复制过程。二路归并排序算法见程序 14-3。

程序 14-3 二路归并排序

```
template<class T>
void MergeSort(T a[], int n)
{ // 使用归并排序算法对 a[0:n-1] 进行排序
  T *b = new T[n];
  int s = 1; // 段的大小
  while (s < n) {
    MergePass(a, b, s, n); // 从 a 归并到 b
    s += s;
    MergePass(b, a, s, n); // 从 b 归并到 a
    s += s;
  }
}
```

为了完成排序代码，首先需要完成函数 MergePass。函数 MergePass（见程序 14-4）仅用来确定欲归并子序列的左端和右端，实际的归并工作由函数 Merge（见程序 14-5）来完成。函数 Merge 要求针对类型 T 定义一个操作符 <=。如果需要排序的数据类型是用户自定义类型，则必须重载操作符 <=。这种设计方法允许我们按元素的任一个域进行排序。重载操作符 <= 的目的是用来比较需要排序的域。

程序 14-4 MergePass 函数

```
template<class T>
void MergePass(T x[], T y[], int s, int n)
{ // 归并大小为 s 的相邻段
  int i = 0;
  while (i <= n - 2 * s) {
    // 归并两个大小为 s 的相邻段
    Merge(x, y, i, i+s-1, i+2*s-1);
    i = i + 2 * s;
  }
  // 剩下不足 2 个元素
```

```

if (i + s < n) Merge(x, y, i, i+s-1, n-1);
else for (int j = i; j <= n-1; j++)
    // 把最后一段复制到 y
    y[j] = x[j];
}

```

程序 14-5 Merge函数

```

template<class T>
void Merge(T c[], T d[], int l, int m, int r)
// 把 c[l:m] 和 c[m:r] 归并到 d[l:r].
{
    int i = l, // 第一段的游标
        j = m+1, // 第二段的游标
        k = l; // 结果的游标

    // 只要在段中存在 i 和 j, 则不断进行归并
    while ((i <= m) && (j <= r))
        if (c[i] <= c[j]) d[k++] = c[i++];
        else d[k++] = c[j++];

    // 考虑余下的部分
    if (i > m) for (int q = j; q <= r; q++)
        d[k++] = c[q];
    else for (int q = i; q <= m; q++)
        d[k++] = c[q];
}

```

## 2. 自然归并排序

自然归并排序 (natural merge sort) 是基本归并排序 (见程序 14-3) 的一种变化。它首先对输入序列中已经存在的有序子序列进行归并。例如, 元素序列  $[4, 8, 3, 7, 1, 5, 6, 2]$  中包含有序的子序列  $[4, 8]$ ,  $[3, 7]$ ,  $[1, 5, 6]$  和  $[2]$ , 这些子序列是按从左至右的顺序对元素表进行扫描而产生的, 若位置  $i$  的元素比位置  $i+1$  的元素大, 则从位置  $i$  进行分割。对于上面这个元素序列, 可找到四个子序列, 子序列 1 和子序列 2 归并可得  $[3, 4, 7, 8]$ , 子序列 3 和子序列 4 归并可得  $[1, 2, 5, 6]$ , 最后, 归并这两个子序列得到  $[1, 2, 3, 4, 5, 6, 7, 8]$ 。因此, 对于上述元素序列, 仅仅使用了两趟归并, 而程序 14-3 从大小为 1 的子序列开始, 需使用三趟归并。作为一个极端的例子, 假设输入的元素序列已经排好序并有  $n$  个元素, 自然归并排序法将准确地识别该序列不必进行归并排序, 但程序 14-3 仍需要进行  $\lceil \log_2 n \rceil$  趟归并。因此自然归并排序将在  $\Theta(n)$  的时间内完成排序。而程序 14-3 将花费  $\Theta(n \log n)$  的时间。

### 14.2.3 快速排序

分而治之方法还可以用于实现另一种完全不同的排序方法, 这种排序法称为快速排序 (quick sort)。在这种方法中,  $n$  个元素被分成三段 (组): 左段 *left*, 右段 *right* 和中段 *middle*。中段仅包含一个元素。左段中各元素都小于等于中段元素, 右段中各元素都大于等于中段元素。因此 *left* 和 *right* 中的元素可以独立排序, 并且不必对 *left* 和 *right* 的排序结果进行合并。 *middle* 中的元素被称为支点 (pivot)。图 14-9 中给出了快速排序的伪代码。



```
//使用快速排序方法对 a[0:n-1]排序
从a[0:n-1]中选择一个元素作为middle，该元素为支点
把余下的元素分割为两段 left 和right，使得left中的元素都小于等于支点，而 right 中的元素都大于等于支点
递归地使用快速排序方法对 left 进行排序
递归地使用快速排序方法对 right 进行排序
所得结果为 left+middle+right
```

图14-9 快速排序的伪代码

考察元素序列 [4,8,3,7,1,5,6,2]。假设选择元素6作为支点，则6位于 *middle*；4, 3, 1, 5, 2 位于 *left*；8, 7位于 *right*。当 *left* 排好后，所得结果为 1, 2, 3, 4, 5；当 *right* 排好后，所得结果为 7, 8。把 *right* 中的元素放在支点元素之后，*left* 中的元素放在支点元素之前，即可得到最终的结果 [1,2,3,4,5,6,7,8]。

把元素序列划分为 *left*、*middle*和*right*可以就地进行（见程序 14-6）。在程序 14-6中，支点总是取位置 *l* 中的元素。也可以采用其他选择方式来提高排序性能，本章稍后部分将给出这样一种选择。

程序14-6 快速排序

```
template<class T>
void QuickSort(T*a, int n)
{
    // 对 a[0:n-1] 进行快速排序
    // 要求 a[n] 必需有最大关键值
    quickSort(a, 0, n-1);
}

template<class T>
void quickSort(T a[], int l, int r)
{
    // 排序 a[l:r], a[r+1] 有大值
    if (l >= r) return;
    int i = l, // 从左至右的游标
        j = r + 1; // 从右到左的游标
    T pivot = a[l];

    // 把左侧 >= pivot 的元素与右侧 <= pivot 的元素进行交换
    while (true) {
        do { // 在左侧寻找 >= pivot 的元素
            i = i + 1;
        } while (a[i] < pivot);
        do { // 在右侧寻找 <= pivot 的元素
            j = j - 1;
        } while (a[j] > pivot);
        if (i >= j) break; // 未发现交换对象
        Swap(a[i], a[j]);
    }

    // 设置 pivot
    a[l] = a[j];
    a[j] = pivot;

    quickSort(a, l, j-1); // 对左段排序
}
```

```
quickSort(a, j+1, r); // 对右段排序
}
```

若把程序14-6中do-while条件内的<号和>号分别修改为≤和≥, 程序14-6仍然正确。实验结果表明使用程序14-6的快速排序代码可以得到比较好的平均性能。为了消除程序中的递归, 必须引入堆栈。不过, 消除最后一个递归调用不须使用堆栈。消除递归调用的工作留作练习(练习13)。

程序14-6所需要的递归栈空间为  $O(n)$ 。若使用堆栈来模拟递归, 则可以把这个空间减少为  $O(\log n)$ 。在模拟过程中, 首先对 *left* 和 *right* 中较小者进行排序, 把较大者的边界放入堆栈中。

在最坏情况下 *left* 总是为空, 快速排序所需的计算时间为  $\Theta(n^2)$ 。在最好情况下, *left* 和 *right* 中的元素数目大致相同, 快速排序的复杂性为  $\Theta(n \log n)$ 。令人吃惊的是, 快速排序的平均复杂性也是  $\Theta(n \log n)$ 。

**定理14-1** 快速排序的平均复杂性为  $\Theta(n \log n)$ 。

**证明** 用  $t(n)$  代表对含有  $n$  个元素的数组进行排序的平均时间。当  $n=1$  时,  $t(n)=d$ ,  $d$  为某一常数。当  $n > 1$  时, 用  $s$  表示左段所含元素的个数。由于在中段中有一个支点元素, 因此右段中元素的个数为  $n-s-1$ 。所以左段和右段的平均排序时间分别为  $t(s)$ ,  $t(n-s-1)$ 。分割数组中元素所需要的时间用  $cn$  表示, 其中  $c$  是一个常数。因为  $s$  有同等机会取  $0 \sim n-1$  中的任何一个值, 故可以得到下面的递归表达式:

$$t(n) \leq cn + \frac{1}{n} \sum_{s=0}^{n-1} [t(s) + t(n-s-1)]$$

可将上式化简为:

$$t(n) = cn + \frac{2}{n} \sum_{s=0}^{n-1} t(s) \leq cn + \frac{4d}{n} + \frac{2}{n} \sum_{s=2}^{n-1} t(s) \quad (14-8)$$

如对 (14-8) 式中的  $n$  使用归纳法, 可得到  $t(n) \leq kn \log_e n$ , 其中  $n > 1$  且  $k=2(c+d)$ ,  $e \sim 2.718$  为自然对数的基底。在归纳开始时首先验证  $n=2$  时公式的正确性。根据公式 (14-8), 可以得到  $t(2) \leq 2c+2d \leq kn \log_e 2$ 。在归纳假设部分, 假定  $t(n) \leq kn \log_e n$  (当  $2 \leq n < m$  时,  $m$  是任意一个比 2 大的整数), 然后需证明  $t(m) \leq km \log_e m$ 。根据公式 (14-8) 和归纳假设, 可以得到:

$$t(m) \leq cm + \frac{4d}{m} + \frac{2}{m} \sum_{s=2}^{m-1} t(s) \leq cm + \frac{4d}{m} + \frac{2k}{m} \sum_{s=2}^{m-1} s \log_e s \quad (14-9)$$

为了进一步证明的需要, 提供以下事实:

- $s \log_e s$  是  $s$  的一个递增函数
- $\int_2^m s \log_e s \, ds < \frac{m^2 \log_e m}{2} - \frac{m^2}{4}$

利用以上事实和公式 (14-9), 可以得到:

$$\begin{aligned} t(m) &< cm + \frac{4d}{m} + \frac{2k}{m} \int_2^m s \log_e s \, ds < cm + \frac{4d}{m} + \frac{2k}{m} \left[ \frac{m^2 \log_e m}{2} - \frac{m^2}{4} \right] \\ &= cm + \frac{4d}{m} + km \log_e m - \frac{km}{2} < km \log_e m \end{aligned}$$

图14-10对本书中所讨论的算法在平均条件下和最坏条件下的复杂性进行了比较。

方 法	最坏复杂性	平均复杂性
冒泡排序	$n^2$	$n^2$
计数排序	$n^2$	$n^2$
插入排序	$n^2$	$n^2$
选择排序	$n^2$	$n^2$
堆排序	$n \log n$	$n \log n$
归并排序	$n \log n$	$n \log n$
快速排序	$n^2$	$n \log n$

图14-10 各种排序算法的比较

中值快速排序 ( median-of-three quick sort ) 是程序 14-6 的一种变化, 这种算法有更好的平均性能。注意到在程序 14-6 中总是选择  $a[1]$  做为支点, 而在这种快速排序算法中, 可以不必使用  $a[1]$  做为支点, 而是取  $\{a[1], a[(1+r)/2], a[r]\}$  中大小居中的那个元素作为支点。例如, 假如有三个元素, 大小分别为 5, 9, 7, 那么取 7 为支点。为了实现中值快速排序算法, 一种最简单的方式就是首先选出中值元素并与  $a[1]$  进行交换, 然后利用程序 14-6 完成排序。如果  $a[r]$  是被选出的中值元素, 那么将  $a[1]$  与  $a[r]$  进行交换, 然后将  $a[1]$  (即原来的  $a[r]$ ) 赋值给程序 14-6 中的变量  $\text{pivot}$ , 之后继续执行程序 14-6 中的其余代码。

$n$	插 入 排 序	堆 排 序	归 并 排 序	快 速 排 序
10	0.000068	0.000126	0.000108	0.000081
20	0.000151	0.000234	0.000222	0.000148
30	0.000267	0.000360	0.000323	0.000224
40	0.000416	0.000502	0.000434	0.000304
50	0.000594	0.000646	0.000541	0.000385
60	0.000809	0.000791	0.000646	0.000468
70	0.001057	0.000946	0.000855	0.000552
80	0.001340	0.001103	0.000971	0.000638
90	0.001626	0.001260	0.001090	0.000726
100	0.001984	0.001423	0.001216	0.000814
200	0.006593	0.003140	0.002498	0.001733
300	0.015934	0.004950	0.004194	0.002693
400	0.026923	0.006593	0.005495	0.003688
500	0.041758	0.008791	0.007143	0.004696
600	0.060440	0.010989	0.008242	0.005495
700	0.080220	0.012637	0.009890	0.006593
800	0.106044	0.015385	0.012088	0.008242
900	0.132418	0.017033	0.012637	0.008791
1000	0.164835	0.019231	0.014835	0.009890

时间(s)

图14-11 各种排序算法的平均时间

图14-11中分别给出了根据实验所得到的归并排序、堆排序、插入排序、快速排序的平均

时间。对于每一个不同的  $n$ ，都随机产生了至少 100 组整数。随机整数的产生是通过反复调用 `stdlib.h` 库中的 `random` 函数来实现的。如果对一组整数进行排序的时间少于 10 个时钟滴答，则继续对其他组整数进行排序，直到所用的时间不低于 10 个时钟滴答。在图 14-11 中的数据包含产生随机整数的时间。对于每一个  $n$ ，在各种排序法中用于产生随机整数及其他开销的时间是相同的。因此，图 14-11 中的数据对于比较各种排序算法是很有用的。当  $n = 100$  时，用图 14-12 中的曲线来显示这些数据。

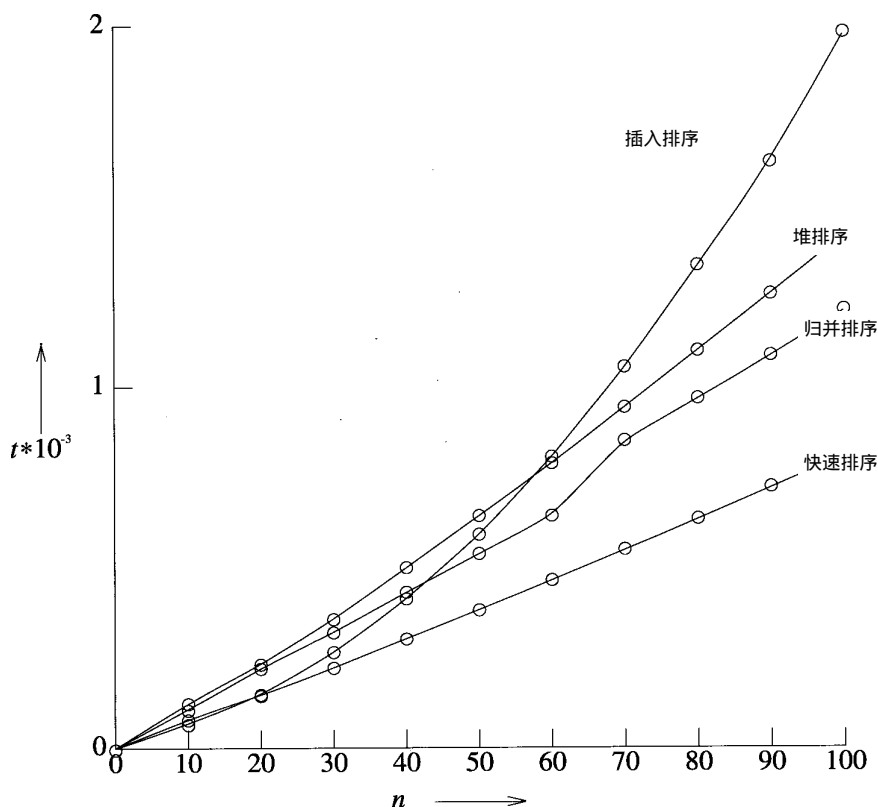


图14-12 各排序算法平均时间的曲线图

如图14-12所示，对于足够大的  $n$ ，快速排序算法要比其他算法效率更高。从图中可以看到快速排序曲线与插入排序曲线的交点横坐标比 20 略小，可通过实验来确定这个交点横坐标的精确值。可以分别用  $n=15, 16, 17, 18, 19$  进行实验，以寻找精确的交点。令精确的交点横坐标为  $n_{\text{Break}}$ 。当  $n \leq n_{\text{Break}}$  时，插入排序的平均性能最佳。当  $n > n_{\text{Break}}$  时，快速排序性能最佳。当  $n > n_{\text{Break}}$  时，把插入排序与快速排序组合为一个排序函数，可以提高快速排序的性能，实现方法是把程序 14-6 中的以下语句：

```
if (l >= r) return;
```

替换为

```
if (r-1 < nBreak) { InsertionSort(a, l, r); return; }
```

这里 `InsertionSort(a, l, r)` 用来对 `a[l:r]` 进行插入排序。测量修改后的快速排序算法的性能留作练习（练习 20）。用更小的值替换  $n_{\text{Break}}$  有可能使性能进一步提高（见练习 20）。

大多数实验表明, 当  $n > c$  时 ( $c$  为某一常数), 在最坏情况下归并排序的性能也是最佳的。而当  $n < c$  时, 在最坏情况下插入排序的性能最佳。通过将插入排序与归并排序混合使用, 可以提高归并排序的性能 (练习 21)。

#### 14.2.4 选择

对于给定的  $n$  个元素的数组  $a[0:n-1]$ , 要求从中找出第  $k$  小的元素。当  $a[0:n-1]$  被排序时, 该元素就是  $a[k-1]$ 。假设  $n=8$ , 每个元素有两个域  $key$  和  $ID$ , 其中  $key$  是一个整数,  $ID$  是一个字符。假设这 8 个元素为  $[(12,a), (4,b), (5,c), (4,d), (5,e), (10,f), (2,g), (20,h)]$ , 排序后得到数组  $[(2,g), (4,d), (4,b), (5,c), (5,e), (10,f), (12,a), (20,h)]$ 。如果  $k=1$ , 返回  $ID$  为  $g$  的元素; 如果  $k=8$ , 返回  $ID$  为  $h$  的元素; 如果  $k=6$ , 返回  $ID$  为  $f$  的元素; 如果  $k=2$ , 返回  $ID$  为  $d$  的元素。实际上, 对最后一种情况, 所得到的结果可能不唯一, 因为排序过程中既可能将  $ID$  为  $d$  的元素排在  $a[1]$ , 也可能将  $ID$  为  $b$  的元素排在  $a[1]$ , 原因是它们具有相同大小的  $key$ , 因而两个元素中的任何一个都有可能被返回。但是无论如何, 如果一个元素在  $k=2$  时被返回, 另一个就必须在  $k=3$  时被返回。

选择问题的一个应用就是寻找中值元素, 此时  $k = \lfloor n/2 \rfloor$ 。中值是一个很有用的统计量, 例如中间工资, 中间年龄, 中间重量。其他  $k$  值也是有用的。例如, 通过寻找第  $n/4, n/2$  和  $3n/4$  这三个元素, 可将人口划分为 4 份。

选择问题可在  $O(n \log n)$  时间内解决, 方法是首先对这  $n$  个元素进行排序 (如使用堆排序式或归并排序), 然后取出  $a[k-1]$  中的元素。若使用快速排序 (如图 14-11 所示), 可以获得更好的平均性能, 尽管该算法有一个比较差的渐近复杂性  $O(n^2)$ 。

可以通过修写程序 14-6 来解决选择问题。如果在执行两个 `while` 循环后支点元素  $a[l]$  被交换到  $a[j]$ , 那么  $a[l]$  是  $a[l:j]$  中的第  $j-l+1$  个元素。如果要寻找的第  $k$  个元素在  $a[l:r]$  中, 并且  $j-l+1$  等于  $k$ , 则答案就是  $a[l]$ ; 如果  $j-l+1 < k$ , 那么寻找的元素是 *right* 中的第  $k-j+1-1$  个元素, 否则要寻找的元素是 *left* 中的第  $k$  个元素。因此, 只需进行 0 次或 1 次递归调用。新代码见程序 14-7。Select 中的递归调用可用 `for` 或 `while` 循环来替代 (练习 25)。

程序 14-7 寻找第  $k$  个元素

```
template<class T>
T Select(T a[], int n, int k)
// 返回a[0:n-1]中第k小的元素
// 假定 a[n] 是一个伪最大元素
if (k < 1 || k > n) throw OutOfBounds();
return select(a, 0, n-1, k);
}

template<class T>
T select(T a[], int l, int r, int k)
// 在 a[l:r]中选择第k小的元素
if (l >= r) return a[l];
int i = l,    // 从左至右的游标
    j = r + 1; // 从右到左的游标
T pivot = a[l];

// 把左侧 >= pivot 的元素与右侧 <= pivot 的元素进行交换
```

```

while (true) {
    do { // 在左侧寻找 >= pivot 的元素
        i = i + 1;
    } while (a[i] < pivot);
    do { // 在右侧寻找 <= pivot 的元素
        j = j - 1;
    } while (a[j] > pivot);
    if (i >= j) break; // 未发现交换对象
    Swap(a[i], a[j]);
}

if (j - l + 1 == k) return pivot;

// 设置pivot
a[l] = a[j];
a[j] = pivot;

// 对一个段进行递归调用
if (j - l + 1 < k)
    return select(a, j+1, r, k-j+l-1);
else return select(a, l, j-1, k);
}

```

程序14-7在最坏情况下的复杂性是  $\Theta(n^2)$ ，此时 *left* 总是为空，而且第 *k* 个元素总是位于 *right*，如果 *left* 和 *right* 总是同样大小或者相差不超过一个元素，那么可以得到以下递归表达式：

$$t(n) \leq \begin{cases} d & n \leq 1 \\ t(\lfloor n/2 \rfloor) + cn & n > 1 \end{cases} \quad (14-10)$$

如果假定 *n* 是2的幂，则可以取消公式 (14-10) 中的向下取整操作符。通过使用迭代方法，可以得到  $t(n) = \Theta(n)$ 。若仔细地选择支点元素，则最坏情况下的时间开销也可以变成  $\Theta(n)$ 。一种选择支点元素的方法是使用“中间的中间 (median-of-median)”规则，该规则首先将数组 *a* 中的 *n* 个元素分成 *n/r* 组，*r* 为某一整常数，除了最后一组外，每组都有 *r* 个元素。然后通过在每组中对 *r* 个元素进行排序来寻找每组中位于中间位置的元素。最后根据所得到的 *n/r* 个中间元素，递归使用选择算法，求得所需要的支点元素。

**例14-6 [中间的中间]** 考察如下情形：*r*=5, *n*=27, 并且 *a*=[2, 6, 8, 1, 4, 10, 20, 6, 22, 11, 9, 8, 4, 3, 7, 8, 16, 11, 10, 8, 2, 14, 15, 1, 12, 5, 4]。这27个元素可以被分为6组 [2,6,8,1,4], [10,20,6,22,11], [9,8,4,3,7], [8,16,11,10,8], [2,14,15,1,12]和[5,4]，每组的中间元素分别为4,11,7,10,12和4。[4,11,7,10,12,4]的中间元素为7。这个中间元素7被取为支点元素。由此可以得到 *left*=[2,6,1,4,6,4,3,2,1,5,4] *middle*=[7], *right*=[8,10,20,22,11,9,8,8,16,11,10,8,14,15,12]。如果要寻找第 *k* 个元素且 *k*<12，则仅仅需要在 *left* 中寻找；如果 *k*=12，则要找的元素就是支点元素；如果 *k*>12，则需要检查 *right* 中的15个元素。在最后一种情况下，需在 *right* 中寻找第 (*k*-12) 个元素。

**定理14-2** 当按“中间的中间”规则选取支点元素时，以下结论为真：

1) 若 *r*=9, 那么当 *n* ≥ 90 时，有  $\max\{|left|, |right|\} \leq 7n/8$ 。

2) 若 $r=5$ ，且 $a$ 中所有元素都不同，那么当 $n \geq 24$ 时，有 $\max\{|left|, |right|\} \leq 3n/4$ 。

证明 这个定理的证明留作练习23。

根据定理14-2和程序14-7可知，如果采用“中间的中间”规则并取 $r=9$ ，则用于寻找第 $k$ 个元素的时间 $t(n)$ 可按如下递归公式来计算：

$$t(n) = \begin{cases} cn \log n & n < 90 \\ t(\lceil n/9 \rceil) + t(\lfloor 7n/8 \rfloor) + cn & n \geq 90 \end{cases} \quad (14-11)$$

在上述递归公式中，假设当 $n < 90$ 时使用复杂性为 $n \log n$ 的求解算法，当 $n \geq 90$ 时，采用“中间的中间”规则进行分而治之求解。利用归纳法可以证明，当 $n \geq 1$ 时有 $t(n) \leq 72cn$ （练习24）。当元素互不相同时，可以使用 $r=5$ 来得到线性时间性能。

#### 14.2.5 距离最近的点对

给定 $n$ 个点 $(x_i, y_i) (1 \leq i \leq n)$ ，要求找出其中距离最近的两个点。两点间的距离公式如下：

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

例14-7 假设在一片金属上钻 $n$ 个大小一样的洞，如果洞太近，金属可能会断。若知道任意两个洞的最小距离，可估计金属断裂的概率。这种最小距离问题实际上也就是距离最近的点对问题。

通过检查所有的 $n(n-1)/2$ 对点，并计算每一对点的距离，可以找出距离最近的一对点。这种方法所需要的时间为 $\Theta(n^2)$ 。我们称这种方法为直接方法。图14-13中给出了分而治之求解算法的伪代码。

该算法对于小的问题采用直接方法求解，而对于大的问题则首先把它划分为两个较小的问题，其中一个问题（称为 $A$ ）的大小为 $\lceil n/2 \rceil$ ，另一个问题（称为 $B$ ）的大小为 $\lceil n/2 \rceil$ 。初始时，最近的点对可能属于如下三种情形之一：1) 两点都在 $A$ 中（即最近的点对落在 $A$ 中）；2) 两点都在 $B$ 中；3) 一点在 $A$ ，一点在 $B$ 。假定根据这三种情况来确定最近点对，则最近点对是所有三种情况中距离最小的一对点。在第一种情况下可对 $A$ 进行递归求解，而在第二种情况下可对 $B$ 进行递归求解。

```

if (n较小) {用直接法寻找最近点对
    Return;}

// n较大
将点集分成大致相等的两个部分A和B
确定A和B中的最近点对
确定一点在A中、另一点在B中的最近点对
从上面得到的三对点中，找出距离最小的一对点

```

图14-13 寻找最近的点对

为了确定第三种情况下的最近点对，需要采用一种不同的方法。这种方法取决于点集是如何被划分成 $A$ 、 $B$ 的。一个合理的划分方法是从 $x_i$ （中间值）处划一条垂线，线左边的点属于 $A$ ，线右边的点属于 $B$ 。位于垂线上的点可在 $A$ 和 $B$ 之间分配，以便满足 $A$ 、 $B$ 的大小。



例14-8 考察图14-14a 中从 $a$ 到 $n$ 的14个点。这些点标绘在图14-14b 中。中点 $x_i=1$ ，垂线 $x=1$ 如图14-14b 中的虚线所示。虚线左边的点(如 $b, c, h, n, i$ )属于 $A$ ，右边的点(如 $a, e, f, j, k, l$ )属于 $B$ 。 $d, g, m$ 落在垂线上，可将其中两个加入 $A$ ，另一个加入 $B$ ，以便 $A, B$ 中包含相同的点数。假设 $d, m$ 加入 $A$ ， $g$ 加入 $B$ 。

点	$a$	$b$	$c$	$d$	$e$	$f$	$g$
$x_i$	2	0.5	0.25	1	3	2	1
$y_i$	2	0.5	1	2	1	0.7	1
点	$h$	$i$	$j$	$k$	$l$	$m$	$n$
$x_i$	0.6	0.9	2	4	1.1	1	0.7
$y_i$	0.8	0.5	1	2	0.5	1.5	2

a)

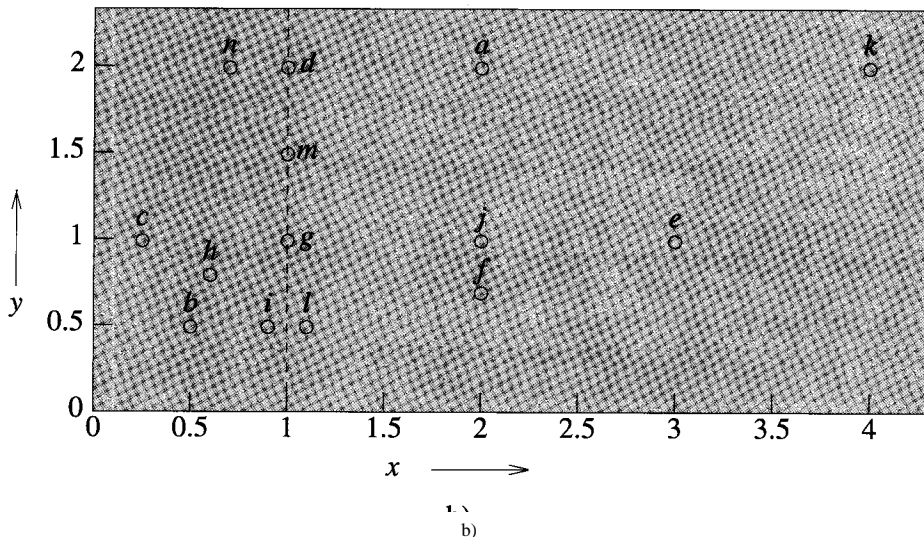


图14-14 14个点

a) 14个点 b) 点的分布

设 $\delta$ 是 $i$ 的最近点对和 $B$ 的最近点对中距离较小的一对点。若第三种情况下的最近点对比 $\delta$ 小。则每一个点距垂线的距离必小于 $\delta$ ，这样，就可以淘汰那些距垂线距离 $\geq \delta$ 的点。图14-15中的虚线是分割线。阴影部分以分割线为中线，宽为 $2\delta$ 。边界线及其以外的点均被淘汰掉，只有阴影中的点被保留下来，以便确定是否存在第三类点对（对应于第三种情况），其距离小于 $\delta$ 。

用 $R_A, R_B$ 分别表示 $A$ 和 $B$ 中剩下的点。如果存在点对 $(p, q)$ ， $p \in A, q \in B$ 且 $p, q$ 的距离小于 $\delta$ ，则 $p \in R_A, q \in R_B$ 。可以通过每次检查 $R_A$ 中一个点来寻找这样的点对。假设考察 $R_A$ 中的 $p$ 点， $p$ 的 $y$ 坐标为 $p.y$ ，那么只需检查 $R_B$ 中满足 $p.y - \delta < q.y < p.y + \delta$ 的 $q$ 点，看是否存在与 $p$ 间距小于 $\delta$ 的点。在图14-16a中给出了包含这种 $q$ 点的 $R_B$ 的范围。因此，只需将 $R_B$ 中位于 $\delta \times 2\delta$ 阴影内的点逐个与 $p$ 配对，以判断 $p$ 是否是距离小于 $\delta$ 的第三类点。这个 $\delta \times 2\delta$ 区域被称为是 $p$ 的比较区（comparing region）。

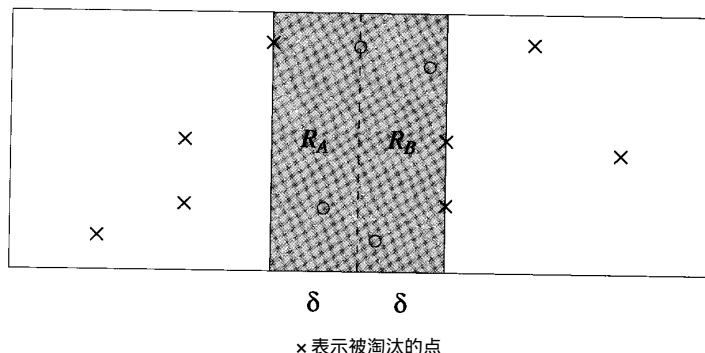


图14-15 淘汰距分割线很远的点

例14-9 考察例14-8中的14个点。A中的最近点对为 $(b, h)$ ，其距离约为0.316。B中最近点对为 $(f, j)$ ，其距离为0.3，因此 $\delta=0.3$ 。当考察是否存在第三类点时，除 $d, g, i, l, m$ 以外的点均被淘汰，因为它们距分割线 $x=1$ 的距离 $\delta$ 。 $R_A=\{d, i, m\}$ ， $R_B=\{g, l\}$ ，由于 $d$ 和 $m$ 的比较区中没有点，只需考察 $i$ 即可。 $i$ 的比较区中仅含点 $l$ 。计算 $i$ 和 $l$ 的距离，发现它小于 $\delta$ ，因此 $(i, l)$ 是最接近的点对。

为了确定一个距离更小的第三类点， $R_A$ 中的每个点最多只需和 $R_B$ 中的6个点比较，如图14-16所示。

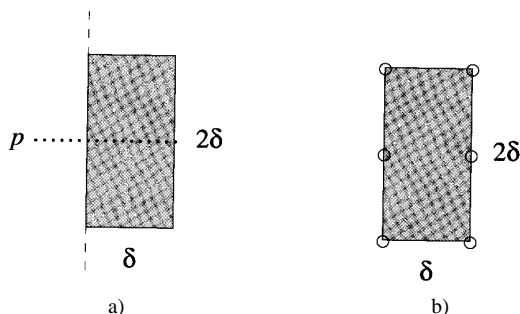


图14-16 p 的比较区

### 1. 选择数据结构

为了实现图14-13的分而治之算法，需要确定什么是“小问题”以及如何表示点。由于集合中少于两点时不存在最近点对，因此必须保证分解过程不会产生少于两点的点集。如果将少于四点的点集做为“小问题”，就可以避免产生少于两点的点集。

每个点可有三个参数：标号， $x$ 坐标， $y$ 坐标。假设标号为整数，每个点可用Point1类（见程序14-8）来表示。为了便于按 $x$ 坐标对各个点排序，可重载操作符 $\leq$ 。归并排序程序如14-3所示。

程序14-8 点类

```
class Point1 {
    friend float dist(const Point1&, const Point1&);
    friend void close(Point1 *, Point2 *, Point2 *, int, int, Point1&, Point1&, float&);
    friend bool closest(Point1 *, int, Point1&, Point1&, float&);
    friend void main();
public:
    int operator<=(Point1 a) const
    {return (x <= a.x);}
private:
    int ID;    // 点的编号
```

```

float x, y; // 点坐标
};

class Point2 {
    friend float dist(const Point2&, const Point2&);
    friend void close(Point1 *, Point2 *, Point2 *, int, int, Point1&, Point1&, float&);
    friend bool closest(Point1 *, int, Point1&, Point1&, float&);
    friend void main();
public:
    int operator<=(Point2 a) const
    {return (y <= a.y);}
private:
    int p;    // 数组 X中相同点的索引
    float x, y; // 点坐标
};

```

所输入的 $n$ 个点可以用数组 $X$ 来表示。假设 $X$ 中的点已按照 $x$ 坐标排序，在分割过程中如果当前考察的点是 $X[l:r]$ ，那么首先计算 $m=(l+r)/2$ ， $X[l:m]$ 中的点属于 $A$ ，剩下的点属于 $B$ 。

计算出 $A$ 和 $B$ 中的最近点对之后，还需要计算 $R_A$ 和 $R_B$ ，然后确定是否存在更近的点对，其中一点属于 $R_A$ ，另一点属于 $R_B$ 。如果点已按 $y$ 坐标排序，那么可以用一种很简单的方式来测试图14-16。按 $y$ 坐标排序的点保存在另一个使用类Point2(见程序14-8)的数组中。注意到在Point2类中，为了便于 $y$ 坐标排序，已重载了操作符 $<=$ 。成员 $p$ 用于指向 $X$ 中的对应点。

确定了必要的数据结构之后，再来看看所要产生的代码。首先定义一个模板函数dist(见程序14-9)来计算点 $a, b$ 之间的距离。 $T$ 可能是Point1或Point2，因此dist必须是Point1和Point2类的友元。

程序14-9 计算两点距离

```

template<class T>
inline float dist(const T& u, const T& v)
{//计算点 u 和 v 之间的距离
    float dx = u.x - v.x;
    float dy = u.y - v.y;
    return sqrt(dx * dx + dy * dy);
}

```

如果点的数目少于两个，则函数closest(见程序14-10)返回false，如果成功时函数返回true。当函数成功时，在参数 $a$ 和 $b$ 中返回距离最近的两个点，在参数 $d$ 中返回距离。代码首先验证至少存在两点，然后使用MergeSort函数(见程序14-3)按 $x$ 坐标对 $X$ 中的点排序。接下来把这些点复制到数组 $Y$ 中并按 $y$ 坐标进行排序。排序完成时，对任一个 $i$ ，有 $Y[i].y \leq Y[i+1].y$ ，并且 $Y[i].p$ 给出了点 $i$ 在 $X$ 中的位置。上述准备工作做完以后，调用函数close(见程序14-11)，该函数实际求解最近点对。

程序14-10 预处理及调用close

```

bool closest(Point1 X[], int n, Point1& a, Point1& b, float& d)

```

```
{// 在n >= 2 个点中寻找最近点对
// 如果少于2个点，则返回 false
// 否则，在 a 和 b中返回距离最近的两个点
if (n < 2) return false;
```

```
// 按 x坐标排序
MergeSort(X,n);
```

```
// 创建一个按y坐标排序的点数组
```

```
Point2 *Y = new Point2 [n];
```

```
for (int i = 0; i < n; i++) {
```

```
    // 将点 i 从 X 复制到 Y
```

```
    Y[i].p = i;
```

```
    Y[i].x = X[i].x;
```

```
    Y[i].y = X[i].y;
```

```
}
```

```
MergeSort(Y,n); // 按 y坐标排序
```

```
// 创建临时数组
```

```
Point2 *Z = new Point2 [n];
```

```
// 寻找最近点对
```

```
close(X,Y,Z,0,n-1,a,b,d);
```

```
// 删除数组并返回
```

```
delete [] Y;
```

```
delete [] Z;
```

```
return true;
```

```
}
```

---

#### 程序14-11 计算最近点对

---

```
void close(Point1 X[], Point2 Y[], Point2 Z[], int l, int r, Point1& a, Point1& b, float& d)
```

```
{//X[l:r] 按x坐标排序
```

```
//Y[l:r] 按y坐标排序
```

```
if (r-l == 1) {// 两个点
```

```
    a = X[l];
```

```
    b = X[r];
```

```
    d = dist(X[l], X[r]);
```

```
    return;}
```

```
if (r-l == 2) {// 三个点
```

```
    // 计算所有点对之间的距离
```

```
    float d1 = dist(X[l], X[l+1]);
```

```
    float d2 = dist(X[l+1], X[r]);
```

```
    float d3 = dist(X[l], X[r]);
```

```
    // 寻找最近点对
```

```
    if (d1 <= d2 && d1 <= d3) {
```

```

a = X[l];
b = X[l+1];
d = d1;
return;}
if (d2 <= d3) {a = X[l+1];
               b = X[r];
               d = d2;}
else {a = X[l];
      b = X[r];
      d = d3;}
return;}

```

//多于三个点，划分为两部分

int m = (l+r)/2; // X[l:m] 在 A 中，余下的在 B 中

// 在 Z[l:m] 和 Z[m+1:r]中创建按y排序的表

int f = l, // Z[l:m]的游标

g = m+1; // Z[m+1:r]的游标

for (int i = l; i <= r; i++)

if (Y[i].p > m) Z[g++] = Y[i];

else Z[f++] = Y[i];

// 对以上两个部分进行求解

close(X,Z,Y,l,m,a,b,d);

float dr;

Point1 ar, br;

close(X,Z,Y,m+1,r,ar,br,dr);

// (a,b) 是两者中较近的对

if (dr < d) {a = ar;

b = br;

d = dr;}

Merge(Z,Y,l,m,r);// 重构 Y

//距离小于d的点放入Z

int k = l; // Z的游标

for (i = l; i <= r; i++)

if (fabs(Y[m].x - Y[i].x) < d) Z[k++] = Y[i];

// 通过检查 Z[l:k-1]中的所有点对，寻找较近的点对

for (i = l; i < k; i++){

for (int j = i+1; j < k && Z[j].y - Z[i].y < d;

j++){

float dp = dist(Z[i], Z[j]);

if (dp < d) {// 较近的点对

d = dp;

a = X[Z[i].p];

b = X[Z[j].p];}

}

```

    }
}

```

函数close (见程序14-11)用来确定 $X[1:r]$ 中的最近点对。假定这些点按 $x$ 坐标排序。在 $Y[1:r]$ 中对这些点按 $y$ 坐标排序。 $Z[1:r]$ 用来存放中间结果。找到最近点对以后,将在 $a, b$ 中返回最近点对,在 $d$ 中返回距离,数组 $Y$ 被恢复为输入状态。函数并未修改数组 $X$ 。

首先考察“小问题”,即少于四个点的点集。因为分割过程不会产生少于两点的数组,因此只需要处理两点和三点的情形。对于这两种情形,可以尝试所有的可能性。当点数超过三个时,通过计算 $m=(1+r)/2$ 把点集分为两组 $A$ 和 $B$ , $X[1:m]$ 属于 $A$ , $X[m+1:r]$ 属于 $B$ 。通过从左至右扫描 $Y$ 中的点以及确定哪些点属于 $A$ ,哪些点属于 $B$ ,可以创建分别与 $A$ 组和 $B$ 组对应的,按 $y$ 坐标排序的 $Z[1:m]$ 和 $Z[m+1:r]$ 。此时 $Y$ 和 $Z$ 的角色互相交换,依次执行两个递归调用来获取 $A$ 和 $B$ 中的最近点对。在两次递归调用返回后,必须保证 $Z$ 不发生改变,但对 $Y$ 则无此要求。不过,仅 $Y[1:r]$ 可能会发生改变。通过合并操作(见程序14-5)可以以 $Z[1:r]$ 重构 $Y[1:r]$ 。

为实现图14-16的策略,首先扫描 $Y[1:r]$ ,并收集距分割线小于 $\delta$ 的点,将这些点存放在 $Z[1:k-1]$ 中。可按如下两种方式来把 $R_A$ 中点 $p$ 与 $p$ 的比较区内的所有点进行配对:1)与 $R_B$ 中 $y$ 坐标 $p.y$ 的点配对;2)与 $y$ 坐标

$p.y$ 的点配对。这可以通过将每个点 $Z[i]$  ( $1 \leq i \leq k$ ,不管该点是在 $R_A$ 还是在 $R_B$ 中)与 $Z[j]$ 配对来实现,其中 $i < j$ 且 $Z[j].y - Z[i].y < \delta$ 。对每一个 $Z[i]$ ,在 $2\delta \times \delta$ 区域内所检查的点如图14-17所示。由于在每个 $2\delta \times \delta$ 子区域内的点至少相距 $\delta$ 。因此每一个子区域中的点数不会超过四个,所以与 $Z[i]$ 配对的点 $Z[j]$ 最多有七个。

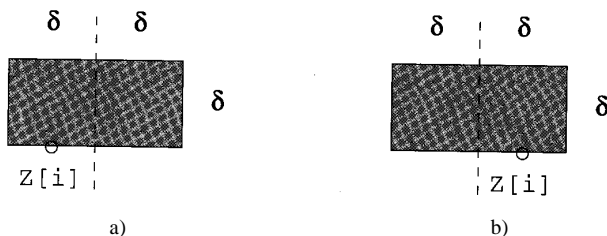


图14-17 与 $Z[i]$ 配对的点的区域

## 2. 复杂性分析

令 $t(n)$ 代表处理 $n$ 个点时,函数close所需要的时间。当 $n < 4$ 时, $t(n)$ 等于某个常数 $d$ 。当 $n \geq 4$ 时,需花费 $\Theta(n)$ 时间来完成以下工作:将点集划分为两个部分,两次递归调用后重构 $Y$ ,淘汰距分割线很远的点,寻找更好的第三类点对。两次递归调用需分别耗时 $t(\lceil n/2 \rceil)$ 和 $t(\lfloor n/2 \rfloor)$ ,因而可得到如下递归式:

$$t(n) = \begin{cases} d & n < 4 \\ t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + cn & n \geq 4 \end{cases}$$

这个递归式与归并排序的递归式完全一样,其结果为 $t(n) = \Theta(n \log n)$ 。另外,函数closest还需耗时 $\Theta(n \log n)$ 来完成如下额外工作:对 $X$ 进行排序,创建 $Y$ 和 $Z$ ,对 $Y$ 进行排序。因此分而治之最近点对求解算法的时间复杂性为 $\Theta(n \log n)$ 。

## 练习

8. 编写一个完整的残缺棋盘问题的求解程序,提供以下模块:欢迎用户使用本程序、输入棋盘大小和残缺方格的位置、输出覆盖后的棋盘。输出棋盘时要着色,共享同一边界的覆盖应着不同的颜色。由于棋盘是平面图,所以最多只需用四种颜色便可为整个棋盘着色。在本练习



中，应尽量使用较少的颜色。

9. 用迭代方法求解公式 (14-7) 的递归式。

10. 编写一个归并排序程序，要求用链表来存储元素。输出结果为排序后的链表。把函数做为 Chain 类 (见程序 3-8) 的一个成员函数。

11. 编写函数 NaturalMergeSort 来实现自然归并排序。其中输入和输出的规定与程序 14-3 相同。

12. 编写一个自然归并排序函数，用来对链表元素进行排序。把函数设计为 Chain 类的一个成员 (见程序 3-8)。

13. 用一个 while 循环来替换程序 14-6 中的最后一个递归调用 quickSort。比较修改后的函数与程序 14-6 的平均运行时间。

14. 重写程序 14-6，使用堆栈来模拟递归。堆栈中只需保存 left 和 right 中较小者的边界。

1) 证明所需要的栈空间大小为  $O(\log n)$ 。

2) 比较程序 14-6 和新代码的平均运行时间。

15. 证明在最坏情况下 QuickSort 的运行时间为  $\Theta(n^2)$ 。

16. 假定在划分 left、middle 和 right 时按照如下方式来进行：若  $n$  为奇数，则 left 与 right 的大小相同；若  $n$  为偶数，则 left 比 right 多一个元素。证明在这种假设条件下，程序 14-6 的时间复杂性为  $\Theta(n \log n)$ 。

17. 证明  $\int_s \log_e s \, ds = \frac{s^2 \log_e s}{2} - \frac{s^2}{4}$ ，并利用该结果证明  $\int_2^m s \log_e s \, ds < \frac{m^2 \log_e m}{2} - \frac{m^2}{4}$ 。

18. 试比较使用“中间的中间”规则与不使用该规则时，程序 14-6 的最坏复杂性和平均复杂性。取  $n=10, 20, \dots, 100, 200, 300, 400, 500, 1000$  及适当的测试数据来进行比较。

19. 采用随机产生的数作为支点元素完成练习 18。

20. 在 14.2.3 节快速排序结束时，我们曾建议将快速排序与插入排序进行结合，结合后的算法实质上仍是快速排序，只是当排序部分小于等于  $\text{ChangeOver} = n\text{Break}$  时执行插入排序。能否通过改变 ChangeOver 而得到更快的算法？为什么？试试不同的 ChangeOver。确定能提供最佳平均性能的 ChangeOver。

21. 设计一个在最差情况下性能最好的排序算法。

1) 比较插入排序、冒泡排序、选择排序、堆排序、归并排序和快速排序在最坏情况下的运行时间。导致插入排序、冒泡排序、选择排序和快速排序出现最坏复杂性的输入数据很容易产生。试编写一个程序，用来产生导致归并排序出现最坏复杂性的输入数据。这个程序本质上是将在  $n$  个排好序的元素“反归并”。对于堆排序，用随机产生的输入序列来估算最坏情况下的时间复杂性。

2) 利用 1) 中的结果设计一个混合排序函数，使其在最坏情况下具有最佳性能。比如混合函数可以只包含归并排序和插入排序。

3) 测试混合排序函数在最坏情况下的运行时间，并与原排序函数进行比较。

4) 用一个简单的图表来列出七种排序函数在最差情况下的运行时间。

22. 当  $n$  是 2 幂时，用迭代方法求解公式 14-8。

\*23. 证明定理 14-2。

\*24. 用归纳法证明，对于公式 (14-11)，当  $n \geq 1$  时有  $t(n) \leq 72cn$ 。

25. 程序 14-7 所需要的递归栈空间为  $O(n)$ 。当用一个 while 或 for 循环来代替递归调用时，可以完全消除这种递归栈空间。根据这种思想重写程序 14-7。比较这两种选择排序函数的运行时间。



26. 重写程序 14-7, 用随机数产生器来选择支点元素。试比较这两种代码的平均性能。

27. 重写程序 14-7, 使用“中间的中间”规则, 其中  $r=9$ 。

28. 为了加快程序 14-11 的执行速度, 可以不执行距离计算公式中的开方运算, 而直接用距离的平方来代替距离, 所得结果是一样的。为此, 程序 14-11 必须做哪些改变? 试通过实验来比较这两种版本的性能。

29. 重写程序 14-11, 把 Point1 作为模板类, 其中 ID 域的类型由用户来决定。

30. 当所有点都在一条直线上时, 编写一个更快的算法来寻找最近的点对。例如, 假设所有点都在一条水平线上。如果这些点根据  $x$  坐标排序, 则最近点对中的两个点必相邻。虽然使用 MergeSort (见程序 14-3) 来实现这种策略时, 算法的复杂性仍然是  $O(n \log n)$ , 但这种算法的额外开销要比程序 14-10 小得多, 因此会运行得更快。

31. 考察最近点对问题。假设初始时不是根据  $x$  坐标来排序, 而是使用 Selcet (见程序 14-7) 来寻找中点  $x_i$ , 以便将点集划分为 A 组和 B 组。

1) 给出按这种思想实现的最近点对问题求解算法的伪代码。

2) 算法的复杂性是多少?

3) 比较新算法与程序 14-11 的运行速度。

### 14.3 解递归方程

许多分而治之算法的复杂性都是由一个递归方程给出, 形式如下:

$$t(n) = \begin{cases} t(1) & n = 1 \\ a * t(n/b) + g(n) & n > 1 \end{cases} \quad (14-12)$$

其中  $a, b$  为已知常数。需假设  $t(1)$  已知, 且  $n$  为  $b$  的幂 (即  $n=b^k$ )。利用迭代原理, 可以证明:

$$t(n) = n^{\log_b a} [t(1) + f(n)] \quad (14-13)$$

其中  $f(n) = \sum_{j=1}^k h(b^j)$   $h(n) = g(n)/n^{\log_b a}$ 。

图 14-18 列出了不同  $h(n)$  时的  $f(n)$  值。根据这张表, 在分析分而治之算法时可以很容易得到  $t(n)$  的值。

考察一些例子。当  $n$  是 2 的幂时, 折半搜索的递归式为:

$$t(n) = \begin{cases} t(1) & n = 1 \\ t(n/2) + c & n > 1 \end{cases}$$

将这个递归式与公式 (14-11) 比较, 可看出  $a=1, b=2, g(n)=c$ , 因而  $\log_b a=0, h(n)=g(n)/n^{\log_b a}=c=c(\log n)^0=\Theta((\log n)^0)$ 。根据图 14-18 可知,  $f(n)=\Theta(\log n)$ , 因而  $t(n)=n^{\log_b a}=(c+\Theta(\log n))=\Theta(\log n)$ 。

对于归并排序, 有  $a=2, b=2, g(n)=cn$ 。因此,  $\log_b a=1, h(n)=g(n)/n=c=\Theta((\log n)^0)$ 。所以  $f(n)=\Theta(\log n)$  且  $t(n)=n(t(1)+\Theta(\log n))=\Theta(n \log n)$ 。

考察另外一个例子, 其递归表达式如下:

$$t(n)=7t(n/2)+18n^2, n \text{ 且为 } 2 \text{ 的幂}$$

该表达式为  $k=1, c=18$  时 Strassen 矩阵乘法的递归表达式 (公式 (14-6))。因  $a=7, b=2, g(n)=18n^2$ , 所以  $\log_b a=\log_2 7 \approx 2.81, h(n)=18n^2/n^{\log_2 7}=18n^{2-\log_2 7}=O(n^r)$ , 其中  $r=2-\log_2 7 < 0$ , 因而  $f(n)=O(1)$ ,

$t(n) = n^{\log_2 7} (t(1) + O(1)) = \Theta(n^{\log_2 7})$ ,  $t(1)$  假设为常数。

最后一个例子, 考察下面的递归式:

$$t(n) = 9t(n/3) + 4n^6, \quad n \geq 3 \text{ 且为 } 3 \text{ 的幂}$$

将这个递归式与式 14-11 比较, 可得到  $a=9$ ,  $b=3$ ,  $g(n)=4n^6$ , 因而  $\log_b a=2$ ,  $h(n)=4n^6/n^2=4n^4 = \Theta(n^4)$ 。根据图 14-12 可知  $f(n)=\Theta(h(n))=\Theta(n^4)$ , 因而  $t(n)=n^2(t(1)+\Theta(n^4))=\Theta(n^6)$ , 其中  $t(1)$  假设为常数。

$h(n)$	$f(n)$
$O(n^r), r < 0$	$O(1)$
$\Theta((\log n)^i), i \geq 0$	$\Theta(((\log n)^{i+1})/(i+1))$
$\Omega(n^r), r > 0$	$\Theta(h(n))$

图 14-18  $f(n)$  与  $h(n)$  的对应关系

## 练习

32. 用迭代原理证明公式 (14-13) 是递归式 (14-12) 的解。

33. 根据图 14-18 中的表求解以下递归式。假定在每种情况下都有  $t(1)=1$ 。

- 1)  $t(n)=10t(n/3)+11n$ ,  $n \geq 3$  且为 3 的幂。
- 2)  $t(n)=10t(n/3)+11n^5$ ,  $n \geq 3$  且为 3 的幂。
- 3)  $t(n)=27t(n/3)+11n^3$ ,  $n \geq 3$  且为 3 的幂。
- 4)  $t(n)=64t(n/4)+10n^3 \log^2 n$ ,  $n \geq 4$  且为 4 的幂。
- 5)  $t(n)=9t(n/2)+n^2 2^n$ ,  $n \geq 2$  且为 2 的幂。
- 6)  $t(n)=3t(n/8)+n^2 2^n \log n$ ,  $n \geq 8$  且为 8 的幂。
- 7)  $t(n)=128t(n/2)+6n$ ,  $n \geq 2$  且为 2 的幂。
- 8)  $t(n)=128t(n/2)+6n^8$ ,  $n \geq 2$  且为 2 的幂。
- 9)  $t(n)=128t(n/2)+2^n/n$ ,  $n \geq 2$  且为 2 的幂。
- 10)  $t(n)=128t(n/2)+\log^3 n$ ,  $n \geq 2$  且为 2 的幂。

## 14.4 复杂性的下限

当且仅当某个问题至少有一个复杂性为  $O(f(n))$  的求解算法时, 存在一个复杂性的上限 (upper bound)  $f(n)$ 。证明一个问题复杂性上限为  $f(n)$  的一种方法是设计一个复杂性为  $O(f(n))$  的算法。对于本书中的每一个算法, 都给出了所解决问题的复杂性上限。如, 在发现 Strassen 矩阵乘法 (例 14.3) 之前, 矩阵乘法的复杂性上限为  $n^3$  (因为程序 2-24 的复杂性为  $\Theta(n^3)$ )。Strassen 算法的发现使复杂性的上限降为  $n^{2.81}$ 。

当且仅当一个问题所有的求解算法的复杂性均为  $\Omega(f(n))$  时, 存在一个复杂性的下限 (lower bound)  $f(n)$ 。为了确定一个问题的复杂性下限  $g(n)$ , 必须证明该问题的每一个求解算法的复杂性均为  $\Omega(g(n))$ 。要得到这样一个结论相当困难, 因为要考察所有可能的求解算法。

对于大多数问题, 可以建立一个基于输入和/或输出数目的简单下限。例如, 对  $n$  个元素进行排序的算法的复杂性为  $\Omega(n \log n)$ , 因为所有的算法对每一个元素都必须检查至少一遍, 否则未检

查的元素可能会排列在错误的位置上。类似地，每一个计算两个  $n \times n$  矩阵乘法的算法都有复杂性 ( $n^2$ )。因为结果矩阵中有  $n^2$  个元素并且产生每个元素所需要的时间为 (1)。只有极少数问题的下限能够找到一个精确的值。

本节将建立本章所介绍的两个分而治之算法的确切下限——寻找  $n$  个元素中的最大值和最小值问题以及排序。问题对于这两个问题，仅限于考察比较算法 (comparison algorithm)。所谓比较算法是指算法的操作主要限于元素比较和元素移动。第 2 章所介绍的最小最大算法以及本章中所介绍的算法都属于比较算法。除箱子排序和基数排序外，本书中所有介绍的其他所有排序算法也都是比较算法。

#### 14.4.1 最小最大问题的下限

程序 14-1 给出了一个寻找  $n$  个元素中最大值与最小值的分而治之函数，该函数执行了  $[3n/2] - 2$  次元素比较。我们将要证明对于该问题的每一个比较算法，都至少需要比较  $[3n/2] - 2$  次。为了证明该结论，假设  $n$  个元素互不相同。这种假设不会影响证明的普遍性，因为不同元素的输入是输入空间的一个子集。另外，每个算法对于有重复元素和没有重复元素的输入都能正确工作。

证明过程中需要使用状态空间方法 (state space method)。这个方法要求首先定义算法的三种状态：起始状态、中间状态和完成状态，并需描述如何从一个状态转换到另一个状态，然后确定从起始状态到完成状态所需的最少转换数。一个算法的起始状态、中间状态和完成状态是一个抽象的概念，不必寻根问底。

对于最小最大问题，算法状态可用元组  $(a, b, c, d)$  来描述， $a$  表示算法需考察的候选的最大和最小元素的个数， $b$  表示不再做为最小候选但仍作为最大候选的元素个数， $c$  是不再做为最大候选但仍做为最小候选的元素个数， $d$  是被确定为即非最大也非最小的元素个数。 $A, B, C, D$  代表上述各种元素的集合。

在最小最大算法启动时，所有  $n$  个元素都是最大与最小元素的候选，状态为  $(n, 0, 0, 0)$ ，当算法结束时， $A$  为空， $B$  和  $C$  中各有 1 个元素， $D$  中有  $n - 2$  个元素，因此完成状态为  $(0, 1, 1, n - 2)$ 。在比较元素的过程中算法状态发生变化。当  $A$  中的两个元素比较完时，较小的元素放入  $C$ ，较大的元素放入  $B$  (根据假设所有元素都不相同，因此不会出现相等的情形)。下面是一种可能的状态转换：

$$(a, b, c, d) \rightarrow (a - 2, b + 1, c + 1, d)$$

其他可能的状态转换如下：

- $B$  中元素比较之后，可能的转换为：

$$(a, b, c, d) \rightarrow (a, b - 1, c, d + 1)$$

- $C$  中元素比较之后，可能的转换为：

$$(a, b, c, d) \rightarrow (a, b, c - 1, d + 1)$$

- $A$  中元素与  $B$  中元素进行比较，可能的转换为：

$$(a, b, c, d) \rightarrow (a - 1, b, c, d + 1) \text{ (} A \text{ 中元素大于 } B \text{ 中元素)}$$

$$(a, b, c, d) \rightarrow (a - 1, b, c + 1, d) \text{ (} A \text{ 中元素小于 } B \text{ 中元素)}$$

- $A$  中元素与  $C$  中元素进行比较，可能的转换为：

$$(a, b, c, d) \rightarrow (a - 1, b, c, d + 1) \text{ (} A \text{ 中元素小于 } C \text{ 中元素)}$$

$(a, b, c, d) \quad (a-1, b+1, c, d)$  ( $A$ 中元素大于 $C$ 中元素)

虽然也可能进行其他比较，但它们不能确保能使状态发生变化。考查上述可能的状态转换，可以发现，当 $n$ 为偶数时，欲从起始状态 $(n, 0, 0, 0)$ 到达完成状态 $(0, 1, 1, n-2)$ ，最快的方式是在 $A$ 中执行 $n/2$ 次比较，在 $B$ 中执行 $n/2-1$ 次比较，在 $C$ 中执行 $n/2-1$ 次比较，总共需比较 $3n/2-2$ 次；当 $n$ 为奇数时，最快的方式是在 $A$ 中执行 $n/2$ 次比较，在 $B$ 中执行 $n/2-1$ 次比较，在 $C$ 中执行 $n/2-1$ 次比较，另有至多两次 $A$ 中剩余元素的比较，总的比较次数为 $\lceil 3n/2 \rceil - 2$ 。

因为没有哪个算法从起始状态到完成状态的比较次数少于 $\lceil 3n/2 \rceil - 2$ ，因此这个数是所有比较算法所需比较次数的下限。所以程序14-1是解决最大最小问题的理想算法。

#### 14.4.2 排序算法的下限

用状态空间定理可以证明对 $n$ 个元素进行排序时，在最坏情况下比较算法的复杂性下限为 $n \log n$ 。对于排序算法，我们把算法的状态定义为仍可能成为输出候选的 $n$ 个元素的排列个数。算法启动时，对应于 $n$ 个元素的所有 $n!$ 种排列都是候选。当算法结束时，只有一种排列保留下来。（假设 $n$ 个元素互不相同。）

当 $a_i$ 与 $a_j$ 比较时，当前候选的排列集合被分为两组：一组满足 $a_i < a_j$ ；另一组满足 $a_i > a_j$ 。因为已假设元素互不相同，所以 $a_i = a_j$ 不存在。例如，假设 $n=3$ ，则首先比较 $a_1$ 和 $a_3$ 。在比较前，所有六种可能的排列都被作为候选输出。若 $a_1 < a_3$ ，则删除 $(a_3, a_1, a_2)$ ， $(a_3, a_2, a_1)$ 和 $(a_2, a_3, a_1)$ ，余下的三种排列继续做为候选输出。

如果当前候选有 $m$ 个，一次比较之后分成两组，其中一组至少包含 $\lceil m/2 \rceil$ 种排列。最坏情况下算法的初始候选有 $n!$ 个，然后降为至少 $n!/2$ ，再降为至少 $n!/4$ ，如此等等，直到只有一个候选为止。这种候选下降的次数最少有 $\lceil \log n! \rceil$ 。

因为 $n! \lceil n/2 \rceil^{\lceil n/2 \rceil - 1} \log n! \approx (n/2 - 1) \log(n/2) = (n \log n)$ 。所以每种排序算法（同时也是比较算法）在最坏情况下，要进行 $(n \log n)$ 次比较。

也可用决策树（decision-tree）来证明下限。在这种证明过程中用树来模拟算法的执行过程。对于树的每个内部节点，算法执行一次比较并根据比较结果移向它的某一孩子。算法在叶节点处终止。图14-19给出了对三个元素 $a[0:2]$ 使用InsertionSort（见程序2-15）排序时的决策树。每个内部节点有一个 $i:j$ 的标志，表示 $a[i]$ 与 $a[j]$ 进行比较。如果 $a[i] < a[j]$ ，算法移向左孩子；如果 $a[i] > a[j]$ ，移向右孩子。因为元素互不相同，所以 $a[i] = a[j]$ 不会发生。叶节点标出了所产生的排序。图14-19中最左路径代表： $a[1] < a[0]$ ， $a[2] < a[0]$ ， $a[2] < a[1]$ ，因此最左叶节点为 $(a[2], a[1], a[0])$ 。

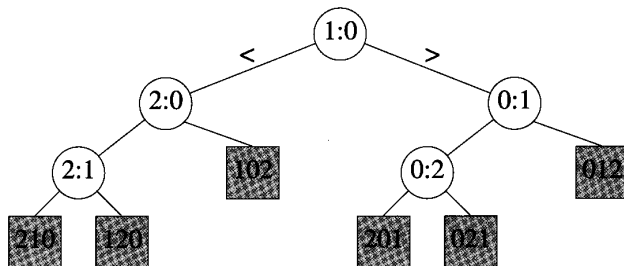


图14-19  $n=3$  时InsertionSort 的决策树

注意到决策树中每个叶节点代表一种唯一的输出排列。由于一个正确的排序算法对于 $n$ 个

输入必须能产生  $n!$  个可能的排列, 因此决策树中至少要有  $n!$  个叶节点。因为一个高度为  $h$  的树至多有  $2^h$  个叶节点, 因此决策树的高度至少为  $\lceil \log_2 n! \rceil = (n \log n)$ , 因而, 每一个比较排序算法在最坏情况下至少要进行  $(n \log n)$  次比较。另外, 由于每个有  $n!$  个叶节点的二叉树的平均高度为  $(n \log n)$ , 因此每个比较排序算法的平均复杂性也是  $(n \log n)$ 。

由前面的证明可以看出, 堆排序、归并排序在最坏情况下有较好的性能 (针对渐进复杂性而言), 堆排序、归并排序、快速排序在平均情况下性能较优。

## 练习

34. 用状态空间方法证明, 要找出  $n$  个元素的最大值, 每一种比较算法都至少要比  $n-1$  次。
35. 证明  $n! \sim \frac{n^n}{e^n} \sqrt{2\pi n}$ 。
36. 画出  $n=4$  时插入排序的决策树。
37. 画出  $n=4$  时归并排序 (见程序 14-3) 的决策树。
38. 令  $a_1, \dots, a_n$  为  $n$  个元素的序列。当且仅当  $a_i > a_j$  ( $i < j$ ) 时,  $a_i$  和  $a_j$  是颠倒的 (inverted), 元素序列中满足颠倒关系的元素对  $(a_i, a_j)$  的个数被称为该元素序列的颠倒数 (inversion number)。
  - 1) 序列 6, 2, 3, 1 的颠倒数是多少?
  - 2)  $n$  个元素的序列中最大的颠倒数是多少?
  - 3) 假设有一种排序算法只比较相邻的元素, 并可能将其交换 (实质上冒泡排序、选择排序和插入排序就是这样做的)。证明这种排序算法必须执行  $(n^2)$  次比较。

## 第15章 动态规划

动态规划是本书介绍的五种算法设计方法中难度最大的一种,它建立在最优原则的基础上。采用动态规划方法,可以优雅而高效地解决许多用贪婪算法或分而治之算法无法解决的问题。在介绍动态规划的原理之后,本章将分别考察动态规划方法在解决背包问题、图象压缩、矩阵乘法链、最短路径、无交叉子集和元件折叠等方面的应用。

### 15.1 算法思想

和贪婪算法一样,在动态规划中,可将一个问题的解决方案视为一系列决策的结果。不同的是,在贪婪算法中,每采用一次贪婪准则便做出一个不可撤回的决策,而在动态规划中,还要考察每个最优决策序列中是否包含一个最优子序列。

**例15-1 [最短路径]** 考察图12-2中的有向图。假设要寻找一条从源节点 $s=1$ 到目的节点 $d=5$ 的最短路径,即选择此路径所经过的各个节点。第一步可选择节点2,3或4。假设选择了节点3,则此时所要求解的问题变成:选择一条从3到5的最短路径。如果3到5的路径不是最短的,则从1开始经过3和5的路径也不会是最短的。例如,若选择的子路径(非最短路径)是3,2,5(耗费为9),则1到5的路径为1,3,2,5(耗费为11),这比选择最短子路径3,4,5而得到的1到5的路径1,3,4,5(耗费为9)耗费更大。

所以在最短路径问题中,假如在的第一次决策时到达了某个节点 $v$ ,那么不管 $v$ 是怎样确定的,此后选择从 $v$ 到 $d$ 的路径时,都必须采用最优策略。

**例15-2 [0/1背包问题]** 考察13.4节的0/1背包问题。如前所述,在该问题中需要决定 $x_1, \dots, x_n$ 的值。假设按 $i=1, 2, \dots, n$ 的次序来确定 $x_i$ 的值。如果置 $x_1=0$ ,则问题转变为相对于其余物品(即物品2,3, ...,  $n$ ),背包容量仍为 $c$ 的背包问题。若置 $x_1=1$ ,问题就变为关于最大背包容量为 $c-w_1$ 的问题。现设 $r = \{c, c-w_1\}$ 为剩余的背包容量。

在第一次决策之后,剩下的问题便是考虑背包容量为 $r$ 时的决策。不管 $x_1$ 是0或是1,  $[x_2, \dots, x_n]$ 必须是第一次决策之后的一个最优方案,如果不是,则会有一个更好的方案 $[y_2, \dots, y_n]$ ,因而 $[x_1, y_2, \dots, y_n]$ 是一个更好的方案。

假设 $n=3, w=[100,14,10], p=[20,18,15], c=116$ 。若设 $x_1=1$ ,则在本次决策之后,可用的背包容量为 $r=116-100=16$ 。 $[x_2, x_3]=[0,1]$ 符合容量限制的条件,所得值为15,但因为 $[x_2, x_3]=[1, 0]$ 同样符合容量条件且所得值为18,因此 $[x_2, x_3]=[0, 1]$ 并非最优策略。即 $x=[1, 0, 1]$ 可改进为 $x=[1, 1, 0]$ 。若设 $x_1=0$ ,则对于剩下的两种物品而言,容量限制条件为116。总之,如果子问题的结果 $[x_2, x_3]$ 不是剩余情况下的一个最优解,则 $[x_1, x_2, x_3]$ 也不会是总体的最优解。

**例15-3 [航费]** 某航线价格表为:从亚特兰大到纽约或芝加哥,或从洛杉矶到亚特兰大的费用为\$100;从芝加哥到纽约票价\$20;而对于路经亚特兰大的旅客,从亚特兰大到芝加哥的费用仅为\$20。从洛杉矶到纽约的航线涉及到对中转机场的选择。如果问题状态的形式为(起点, 终点),那么在选择从洛杉矶到亚特兰大后,问题的状态变为(亚特兰大, 纽约)。从亚特兰大到纽约的最便宜航线是从亚特兰大直飞纽约,票价\$100。而使用直飞方式时,从洛杉矶到纽约



的花费为\$200。不过，从洛杉矶到纽约的最便宜航线为洛杉矶-亚特兰大-芝加哥-纽约，其总花费为\$140（在处理局部最优路径亚特兰大到纽约过程中选择了最低花费的路径：亚特兰大-芝加哥-纽约）。

如果用三维数组（tag，起点，终点）表示问题状态，其中tag为0表示转飞，tag为1表示其他情形，那么在到达亚特兰大后，状态的三维数组将变为（0，亚特兰大，纽约），它对应的最优路径是经由芝加哥的那条路径。

当最优决策序列中包含最优决策子序列时，可建立动态规划递归方程（dynamic-programming recurrence equation），它可以帮助我们高效地解决问题。

**例15-4 [0/1 背包]** 在例15-2的0/1背包问题中，最优决策序列由最优决策子序列组成。假设 $f(i,y)$ 表示例15-2中剩余容量为 $y$ ，剩余物品为 $i, i+1, \dots, n$ 时的最优解的值，即：

$$f(n,y) = \begin{cases} p_n & y \geq w_n \\ 0 & 0 \leq y < w_n \end{cases} \quad (15-1)$$

和

$$f(i,y) = \begin{cases} \max\{f(i+1,y), f(i+1,y-w_i) + p_i\} & y \geq w_i \\ f(i+1,y) & 0 \leq y < w_i \end{cases} \quad (15-2)$$

利用最优序列由最优子序列构成的结论，可得到 $f$ 的递归式。 $f(1,c)$ 是初始时背包问题的最优解。可使用（15-2）式通过递归或迭代来求解 $f(1,c)$ 。从 $f(n,*)$ 开始迭代， $f(n,*)$ 由（15-1）式得出，然后由（15-2）式递归计算 $f(i,*)$ （ $i=n-1, n-2, \dots, 2$ ），最后由（15-2）式得出 $f(1,c)$ 。

对于例15-2，若 $0 \leq y < 10$ ，则 $f(3,y)=0$ ；若 $y \geq 10$ ， $f(3,y)=15$ 。利用递归式（15-2），可得 $f(2,y)=0$ （ $0 \leq y < 10$ ）； $f(2,y)=15$ （ $10 \leq y < 14$ ）； $f(2,y)=18$ （ $14 \leq y < 24$ ）和 $f(2,y)=33$ （ $y \geq 24$ ）。因此最优解 $f(1,116)=\max\{f(2,116), f(2,116-w_1)+p_1\}=\max\{f(2,116), f(2,16)+20\}=\max\{33, 38\}=38$ 。

现在计算 $x_i$ 值，步骤如下：若 $f(1,c)=f(2,c)$ ，则 $x_1=0$ ，否则 $x_1=1$ 。接下来需从剩余容量 $c-w_1$ 中寻求最优解，用 $f(2,c-w_1)$ 表示最优解。依此类推，可得到所有的 $x_i$ （ $i=1 \dots n$ ）值。

在该例中，可得出 $f(2,116)=33 < f(1,116)$ ，所以 $x_1=1$ 。接着利用返回值 $38-p_1=18$ 计算 $x_2$ 及 $x_3$ ，此时 $r=116-w_1=16$ ，又由 $f(2,16)=18$ ，得 $f(3,16)=14 < f(2,16)$ ，因此 $x_2=1$ ，此时 $r=16-w_2=2$ ，所以 $f(3,2)=0$ ，即得 $x_3=0$ 。

动态规划方法采用最优原则（principle of optimality）来建立用于计算最优解的递归式。所谓最优原则即不管前面的策略如何，此后的决策必须是基于当前状态（由上一次决策产生）的最优决策。由于对于有些问题的某些递归式来说并不一定能保证最优原则，因此在求解问题时有必要对它进行验证。若不能保持最优原则，则不可应用动态规划方法。在得到最优解的递归式之后，需要执行回溯（traceback）以构造最优解。

编写一个简单的递归程序来求解动态规划递归方程是一件很诱人的事。然而，正如我们将在下文看到的，如果不努力地去避免重复计算，递归程序的复杂性将非常可观。如果在递归程序设计中解决了重复计算问题时，复杂性将急剧下降。动态规划递归方程也可用迭代方式来求解，这时很自然地避免了重复计算。尽管迭代程序与避免重复计算的递归程序有相同的复杂性，但迭代程序不需要附加的递归栈空间，因此将比避免重复计算的递归程序更快。



## 15.2 应用

## 15.2.1 0/1 背包问题

## 1. 递归策略

在例15-4中已建立了背包问题的动态规划递归方程，求解递归式 (15-2) 的一个很自然的方法便是使用程序15-1中的递归算法。该模块假设  $p$ 、 $w$  和  $n$  为输入，且  $p$  为整型， $F(1,c)$  返回  $f(1,c)$  值。

程序15-1 背包问题的递归函数

```
int F(int i, int y)
// 返回 f(i,y).
if (i == n) return (y < w[n]) ? 0 : p[n];
if (y < w[i]) return F(i+1,y);
return max(F(i+1,y), F(i+1,y-w[i]) + p[i]);
}
```

程序15-1的时间复杂性  $t(n)$  满足： $t(1)=a$ ； $t(n) = 2t(n-1) + b$  ( $n > 1$ )，其中  $a$ 、 $b$  为常数。通过求解可得  $t(n)=O(2^n)$ 。

例15-5 设  $n=5$ ， $p=[6,3,5,4,6]$ ， $w=[2,2,6,5,4]$  且  $c=10$ ，求  $f(1,10)$ 。为了确定  $f(1,10)$ ，调用函数  $F(1,10)$ 。递归调用的关系如图15-1的树型结构所示。每个节点用  $y$  值来标记。对于第  $j$  层的节点有  $i=j$ ，因此根节点表示  $F(1,10)$ ，而它有左孩子和右孩子，分别对应  $F(2,10)$  和  $F(2,8)$ 。总共执行了28次递归调用。但我们注意到，其中可能含有重复前面工作的节点，如  $f(3,8)$  计算过两次，相同情况的还有  $f(4,8)$ 、 $f(4,6)$ 、 $f(4,2)$ 、 $f(5,8)$ 、 $f(5,6)$ 、 $f(5,3)$ 、 $f(5,2)$  和  $f(5,1)$ 。如果保留以前的计算结果，则可将节点数减至19，因为可以丢弃图中的阴影节点。

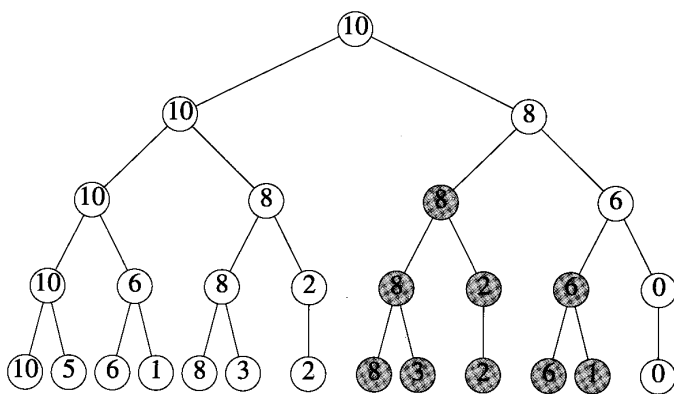


图15-1 递归调用树

正如在例15-5中所看到的，程序15-1做了一些不必要的工作。为了避免  $f(i,y)$  的重复计算，必须定义一个用于保留已被计算出的  $f(i,y)$  值的表格  $L$ ，该表格的元素是三元组  $(i,y,f(i,y))$ 。在计算每一个  $f(i,y)$  之前，应检查表  $L$  中是否已包含一个三元组  $(i,y,*)$ ，其中  $*$  表示任意值。如果已包含，则从该表中取出  $f(i,y)$  的值，否则，对  $f(i,y)$  进行计算并将计算所得的三元组  $(i,y,f(i,y))$  加入

表 $L$ 。 $L$ 既可以用散列（见7.4节）的形式存储，也可用二叉搜索树（见11章）的形式存储。

## 2. 权为整数的迭代方法

当权为整数时，可设计一个相当简单的算法（见程序15-2）来求解 $f(1, c)$ 。该算法基于例15-4所给出的策略，因此每个 $f(i, y)$ 只计算一次。程序15-2用二维数组 $f[i][y]$ 来保存各 $f$ 的值。而回溯函数Traceback用于确定由程序15-2所产生的 $x_i$ 值。

函数Knapsack的复杂性为 $\Theta(nc)$ ，而Traceback的复杂性为 $\Theta(n)$ 。

程序15-2  $f$  和 $x$  的迭代计算

```
template<class T>
void Knapsack(T p[], int w[], int c, int n, T** f)
{// 对于所有 $i$ 和 $y$ 计算 $f[i][y]$ 

    // 初始化  $f[n][y]$ 
    for (int y = 0; y <= yMax; y++)
        f[n][y] = 0;
    for (int y = w[n]; y <= c; y++)
        f[n][y] = p[n];

    // 计算剩下的 $f$ 
    for (int i = n - 1; i > 1; i--) {
        for (int y = 0; y <= yMax; y++)
            f[i][y] = f[i+1][y];
        for (int y = w[i]; y <= c; y++)
            f[i][y] = max(f[i+1][y], f[i+1][y-w[i]] + p[i]);
    }
    f[1][c] = f[2][c];
    if (c >= w[1])
        f[1][c] = max(f[1][c], f[2][c-w[1]] + p[1]);
}

template<class T>
void Traceback(T **f, int w[], int c, int n, int x[])
{// 计算 $x$ 
    for (int i = 1; i < n; i++)
        if (f[i][c] == f[i+1][c]) x[i] = 0;
        else {x[i] = 1;
              c -= w[i];}
    x[n] = (f[n][c]) ? 1 : 0;
}
```

## 3. 元组方法（选读）

程序15-2有两个缺点：1) 要求权为整数；2) 当背包容量 $c$ 很大时，程序15-2的速度慢于程序15-1。一般情况下，若 $c > 2^n$ ，程序15-2的复杂性为 $(n2^n)$ 。可利用元组的方法来克服上述两个缺点。在元组方法中，对于每个 $i$ ， $f(i, y)$ 都以数对 $(y, f(i, y))$ 的形式按 $y$ 的递增次序存储于表

$P(i)$ 中。同时,由于 $f(i, y)$ 是 $y$ 的非递减函数,因此 $P(i)$ 中各数对 $(y, f(i, y))$ 也是按 $f(i, y)$ 的递增次序排列的。

例15-6 条件同例15-5。对 $f$ 的计算如图15-2所示。当 $i=5$ 时, $f$ 由数对集合 $P(5)=[(0,0),(4,6)]$ 表示。而 $P(4)$ 、 $P(3)$ 和 $P(2)$ 分别为 $[(0,0),(4,6),(9,10)]$ 、 $[(0,0),(4,6),(9,10),(10,11)]$ 和 $[(0,0),(2,3),(4,6),(6,9),(9,10),(10,11)]$ 。

为求 $f(1,10)$ ,利用式(15-2)得 $f(1,10)=\max\{f(2,10), f(2,8)+p_1\}$ 。由 $P(2)$ 得 $f(2,10)=11$ 、 $f(2,8)=9$ ( $f(2,8)=9$ 来自数对 $(6,9)$ ),因此 $f(1,10)=\max\{11,15\}=15$ 。

现在来求 $x_1$ 的值,因为 $f(1,10)=f(2,6)+p_1$ ,所以 $x_1=1$ ;由 $f(2,6)=f(3,6-w_2)+p_2=f(3,4)+p_2$ ,得 $x_2=1$ ;由 $f(3,4)=f(4,4)=f(5,4)$ 得 $x_3=x_4=0$ ;最后,因 $f(5,4)=0$ 得 $x_5=1$ 。

$i$	$y$										
	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6	6	6	6	10	10
3	0	0	0	0	6	6	6	6	6	10	11
2	0	0	3	3	6	6	9	9	9	10	11

图15-2 例15-6中的 $f$ 值

检查每个 $P(i)$ 中的数对,可以发现每对 $(y, f(i, y))$ 对应于变量 $x_1, \dots, x_n$ 的0/1赋值的不同组合。设 $(a, b)$ 和 $(c, d)$ 是对应于两组不同 $x_1, \dots, x_n$ 的0/1赋值,若 $a < c$ 且 $b < d$ ,则 $(a, b)$ 受 $(c, d)$ 支配。被支配者不必加入 $P(i)$ 中。若在相同的数对中有两个或更多的赋值,则只有一个放入 $P(i)$ 。

假设 $w_n < C$ ,  $P(n)=[(0,0), (w_n, p_n)]$ ,  $P(n)$ 中对应于 $x_n$ 的两个数对分别等于0和1。对于每个 $i$ ,  $P(i)$ 可由 $P(i+1)$ 得出。首先,要计算数对的有序集合 $Q$ ,使得当且仅当 $w_i \leq s$ 且 $(s-w_i, t-p_i)$ 为 $P(i+1)$ 中的一个数对时,  $(s, t)$ 为 $Q$ 中的一个数对。现在 $Q$ 中包含 $x_i=1$ 时的数对集,而 $P(i+1)$ 对应于 $x_i=0$ 的数对集。接下来,合并 $Q$ 和 $P(i+1)$ 并删除受支配者和重复值即可得到 $P(i)$ 。

例15-7 各数据同例15-6。  $P(5)=[(0,0),(4,6)]$ ,因此 $Q=[(5,4),(9,10)]$ 。现在要将 $P(5)$ 和 $Q$ 合并得到 $P(4)$ 。因 $(5,4)$ 受 $(4,6)$ 支配,可删除 $(5,4)$ ,所以 $P(4)=[(0,0),(4,6),(9,10)]$ 。接着计算 $P(3)$ ,首先由 $P(4)$ 得 $Q=[(6,5),(10,11)]$ ,然后又由合并方法得 $P(3)=[(0,0),(4,6),(9,10),(10,11)]$ 。最后计算 $P(2)$ :由 $P(3)$ 得 $Q=[(2,3),(6,9)]$ ,  $P(3)$ 与 $Q$ 合并得 $P(2)=[(0,0),(2,3),(4,6),(6,9),(9,10),(10,11)]$ 。

因为每个 $P(i)$ 中的数对对应于 $x_1, \dots, x_n$ 的不同0/1赋值,因此 $P(i)$ 中的数对不会超过 $2^{n-i+1}$ 个。计算 $P(i)$ 时,计算 $Q$ 需消耗 $\Theta(|P(i+1)|)$ 的时间,合并 $P(i+1)$ 和 $Q$ 同样需要 $\Theta(|P(i+1)|)$ 的时间。计算所有 $P(i)$ 时所需要的总时间为: $\Theta(\sum_{i=2}^n |P(i+1)|) = O(2^n)$ 。当 $n$ 为整数时,  $|P(i)| \leq c+1$ ,此时复杂性为 $O(\min\{nc, 2^n\})$ 。

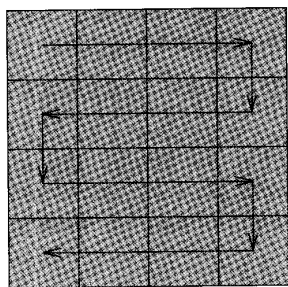
### 15.2.2 图像压缩

如6.4.3节定义的,数字化图像是 $m \times m$ 的像素阵列。假定每个像素有一个0~255的灰度值。因此存储一个像素至多需8位。若每个像素存储都用最大位8位,则总的存储空间为 $8m^2$ 位。为了减少存储空间,我们将采用变长模式(variable bit scheme),即不同像素用不同位数来存储。

像素值为0和1时只需1位存储空间;值2、3各需2位;值4、5、6和7各需3位;以此类推,

使用变长模式的步骤如下：

1) 图像线性化 根据图15-3a 中的折线将  $m \times m$  维图像转换为  $1 \times m^2$  维矩阵。



a)

10	9	12	40
12	15	35	50
8	10	9	15
240	160	130	11

b)

图15-3 数字图像

a) 泡形的行主次序 b) 灰度值

2) 分段 将像素组分成若干个段，分段原则是：每段中的像素位数相同。每个段是相邻像素的集合且每段最多含256个像素，因此，若相同位数的像素超过256个的话，则用两个以上的段表示。

3) 创建文件 创建三个文件：*SegmentLength*、*BitsPerPixel* 和 *Pixels*。第一个文件包含在2)中所建的段的长度(减1)，文件中各项均为8位长。文件 *BitsPerPixel* 给出了各段中每个像素的存储位数(减1)，文件中各项均为3位。文件 *Pixels* 则是以变长格式存储的像素的二进制串。

4) 压缩文件 压缩在3)中所建立的文件，以减少空间需求。

上述压缩方法的效率(用所得压缩率表示)很大程度上取决于长段的出现频率。

例15-8 考察图15-3b 的  $4 \times 4$  图像。按照蛇形的行主次序，灰度值依次为10, 9, 12, 40, 50, 35, 15, 12, 8, 10, 9, 15, 11, 130, 160和240。各像素所需的位数分别为4, 4, 4, 6, 6, 6, 4, 4, 4, 4, 4, 4, 4, 8, 8和8，按等长的条件将像素分段，可以得到4个段[10, 9, 12]、[40, 50, 35]、[15, 12, 8, 10, 9, 15, 11] 和 [130, 160, 240] 因此，文件 *SegmentLength* 为2, 2, 6, 2；文件 *BitsPerSegment* 的内容为3, 5, 3, 7；文件 *Pixels* 包含了按蛇形行主次序排列的16个灰度值，其中头三个各用4位存储，接下来三个各用6位，再接下来的七个各用4位，最后三个各用8位存储。因此存储单元中前30位存储了前六个像素：

1010 1001 1100 111000 110010 100011

这三个文件需要的存储空间分别为：文件 *SegmentLength* 需32位；*BitsPerSegment* 需12位；*Pixels* 需82位，共需126位。而如果每个像素都用8位存储，则存储空间需  $8 \times 16 = 128$  位，因而在本例图像中，节省了2位的空间。

假设在2)之后，产生了  $n$  个段。段标题(segment header)用于存储段的长度以及该段中每个像素所占用的位数。每个段标题需11位。现假设  $l_i$  和  $b_i$  分别表示第  $i$  段的段长和该段每个像素的长度，则存储第  $i$  段像素所需要的空间为  $l_i * b_i$ 。在2)中所得的三个文件的总存储空间为  $11n + \sum_{i=1}^n l_i b_i$ 。可通过将某些相邻段合并的方式来减少空间消耗。如当段  $i$  和  $i+1$  被合并时，合并后的段长应为  $l_i + l_{i+1}$ 。此时每个像素的存储位数为  $\max\{b_i, b_{i+1}\}$  位。尽管这种技术增加了文件 *Pixels* 的空间消耗，但同时也减少了一个段标题的空间。

例15-9 如果将例15-8中的第1段和第2段合并，合并后，文件 *SegmentLength* 变为5, 6, 2, *BitsPerSegment* 变为5, 3, 7。而文件 *Pixels* 的前36位存储的是合并后的第一段：

001010 001001 001100 111000 110010 100011

其余的像素（例15-8第3段）没有改变。因为减少了1个段标题，文件 *SegmentLength* 和 *BitsPerPixel* 的空间消耗共减少了11位，而文件 *Pixels* 的空间增加6位，因此总共节约的空间为5位，空间总消耗为121位。

我们希望能设计一种算法，使得在产生  $n$  个段之后，能对相邻段进行合并，以便产生一个具有最小空间需求的新的段集合。在合并相邻段之后，可利用诸如 LZW法（见7.5节）和霍夫曼编码（见9.5.3节）等其他技术来进一步压缩这三个文件。

令  $s_q$  为前  $q$  个段的最优合并所需要的空间。定义  $s_0=0$ 。考虑第  $i$  段 ( $i>0$ )，假如在最优合并  $C$  中，第  $i$  段与第  $i-1, i-2, \dots, i-r+1$  段相合并，而不包括第  $i-r$  段。合并  $C$  所需要的空间消耗等于：

第1段到第  $i-r$  段所需空间 +  $lsum(i-r+1, i) * bmax(i-r+1, i) + 11$

其中  $lsum(a, b) = \sum_{j=a}^b l_j$ ,  $bmax(a, b) = \max\{b_a, \dots, b_b\}$ 。假如在  $C$  中第1段到第  $i-r$  段的合并不是最优合并，那么需要对合并进行修改，以使其具有更小的空间需求。因此还必须对段1到段  $i-r$  进行最优合并，也即保证最优原则得以维持。故  $C$  的空间消耗为：

$$s_i = s_{i-r} + lsum(i-r+1, i) * bmax(i-r+1, i) + 11$$

$r$  的值介于1到  $i$  之间，其中要求  $lsum$  不超过256(因为段长限制在256之内)。尽管我们不知道如何选择  $r$ ，但我们知道，由于  $C$  具有最小的空间需求，因此在所有选择中， $r$  必须产生最小的空间需求。因此可得递归式：

$$s_i = \min_{\substack{1 \leq k \leq i \\ lsum(i-k+1, i) \leq 256}} \{s_{i-k} + lsum(i-k+1, i) * bmax(i-k+1, i)\} + 11 \quad (15-3)$$

假定  $kay_i$  表示取得最小值时  $k$  的值， $s_n$  为  $n$  段的最优合并所需要的空间，因而一个最优合并可用  $kay$  的值构造出来。

例15-10 假定在2)中得到五个段，它们的长度为[6, 3, 10, 2, 3]，像素位数为[1, 2, 3, 2, 1]，要用公式(15-3)计算  $s_n$ ，必须先求出  $s_{n-1}, \dots, s_0$  的值。 $s_0$  为0，现计算  $s_1$ ：

$$s_1 = s_0 + l_1 * b_1 + 11 = 17$$

$$kay_1 = 1$$

$s_2$  由下式得出：

$$s_2 = \min\{s_1 + l_2 * b_2, s_0 + (l_1 + l_2) * \max\{b_1, b_2\}\} + 11 = \min\{17 + 6 * 0 + 9 * 2\} + 11 = 29$$

$$kay_2 = 2$$

以此类推，可得  $s_1 \dots s_5 = [17, 29, 67, 73, 82]$ ， $kay_1 \dots kay_5 = [1, 2, 2, 3, 4]$ 。

因为  $s_5 = 82$ ，所以最优空间合并需82位的空间。可由  $kay_5$  导出本合并的方式，过程如下：因为  $kay_5 = 4$ ，所以  $s_5$  是由公式(15-3)在  $k=4$  时取得的，因而最优合并包括：段1到段  $(5-4)=1$  的最优合并以及段2, 3, 4和5的合并。最后只剩下两个段：段1以及段2到段5的合并段。

### 1. 递归方法

用递归式(15-3)可以递归地算出  $s_i$  和  $kay_i$ 。程序15-3为递归式的计算代码。 $l, b$ ，和  $kay$  是一维的全局整型数组， $L$  是段长限制(256)， $header$  为段标题所需的空(11)。调用  $S(n)$  返回

$s_n$  的值且同时得出  $kay$  值。调用  $\text{Traceback}(kay, n)$  可得到最优合并。

现讨论程序 15-3 的复杂性。  $t(0)=c$  ( $c$  为一个常数) :  $t(n) \leq \sum_{j=\max\{0, n-256\}}^{n-1} t(j) + n \quad (n > 0)$ ,

因此利用递归的方法可得  $t(n)=O(2^n)$ 。  $\text{Traceback}$  的复杂性为  $\Theta(n)$ 。

程序 15-3 递归计算  $s, kay$  及最优合并

```
int S(int i)
{ // 返回 S(i) 并计算 kay[i]
  if (i == 0) return 0;
  // k = 1 时, 根据公式 (15-3) 计算最小值
  int lsum = l[i], bmax = b[i];
  int s = S(i-1) + lsum * bmax;
  kay[i] = 1;
  // 对其余的 k 计算最小值并求取最小值
  for (int k = 2; k <= i && lsum + l[i-k+1] <= L; k++) {
    lsum += l[i-k+1];
    if (bmax < b[i-k+1]) bmax = b[i-k+1];
    int t = S(i-k);
    if (s > t + lsum * bmax) {
      s = t + lsum * bmax;
      kay[i] = k;
    }
  }

  return s + header;
}

void Traceback(int kay[], int n)
{ // 合并段
  if (n == 0) return;
  Traceback(kay, n-kay[n]);
  cout << "New segment begins at " << (n - kay[n] + 1) << endl;
}
```

## 2. 无重复计算的递归方法

通过避免重复计算  $s_i$ , 可将函数  $S$  的复杂性减少到  $\Theta(n)$ 。注意这里只有  $n$  个不同的  $s_i$ 。

例 15-11 再考察例 15-10 中五个段的例子。当计算  $s_5$  时, 先通过递归调用来计算  $s_4, \dots, s_0$ 。计算  $s_4$  时, 通过递归调用计算  $s_3, \dots, s_0$ , 因此  $s_4$  只计算了一次, 而  $s_3$  计算了两次, 每一次计算  $s_3$  要计算一次  $s_2$ , 因此  $s_2$  共计算了四次, 而  $s_1$  重复计算了 16 次!

可利用一个数组  $s$  来保存先前计算过的  $s_i$  以避免重复计算。改进后的代码见程序 15-4, 其中  $s$  为初值为 0 的全局整型数组。

程序 15-4 避免重复计算的递归算法

```
int S(int i)
{ // 计算 S(i) 和 kay[i]
  // 避免重复计算
```

```

if (i == 0) return 0;
if (s[i] > 0) return s[i]; //已计算完
//计算 s[i]
//首先根据公式 (15-3) 计算k = 1时最小值
int lsum = l[i], bmax = b[i];
s[i] = S(i-1) + lsum * bmax;
kay[i] = 1;
//对剩余的k计算最小值并更新
for (int k = 2; k <= i && lsum + l[i-k+1] <= L; k++) {
    lsum += l[i-k+1];
    if (bmax < b[i-k+1]) bmax = b[i-k+1];
    int t = S(i-k);
    if (s[i] > t + lsum * bmax) {
        s[i] = t + lsum * bmax;
        kay[i] = k;
    }
}

s[i] += header;
return s[i];
}

```

为了确定程序 15-4 的时间复杂性，我们将使用分期计算模式 (amortization scheme)。在该模式中，总时间被分解为若干个不同项，通过计算各项的时间然后求和来获得总时间。当计算  $s_i$  时，若  $s_j$  还未算出，则把调用  $S(j)$  的消耗计入  $s_j$ ；若  $s_j$  已算出，则把  $S(j)$  的消耗计入  $s_i$  (这里  $s_j$  依次把计算新  $s_q$  的消耗转移至每个  $s_q$ )。程序 15-4 的其他消耗也被计入  $s_p$ 。因为  $L$  是 256 之内的常数且每个  $l_i$  至少为 1，所以程序 15-4 的其他消耗为  $\Theta(1)$ ，即计入每个  $s_i$  的量是一个常数，且  $s_i$  数目为  $n$ ，因而总工作量为  $\Theta(n)$ 。

### 3. 迭代方法

倘若用式 (15-3) 依序计算  $s_1, \dots, s_n$ ，便可得到一个复杂性为  $\Theta(n)$  的迭代方法。在该方法中，在  $s_i$  计算之前， $s_j$  必须已计算好。该方法的代码见程序 15-5，其中仍利用函数 Traceback (见程序 15-3) 来获得最优合并。

程序 15-5 迭代计算  $s$  和  $kay$

```

void Vbits (int l[], int b[], int n, int s[], int kay[])
{ //计算 s[i] 和 kay[i]
    int L = 256, header = 11;
    s[0] = 0;
    //根据式 (15-3) 计算 s[i]
    for (int i = 1; i <= n; i++) {
        // k = 1 时, 计算最小值
        int lsum = l[i],
            bmax = b[i];
        s[i] = s[i-1] + lsum * bmax;
        kay[i] = 1;
        //对剩余的 k 计算最小值并更新
        for (int k=2; k<= i && lsum+l[i-k+1]<= L; k++) {
            lsum += l[i-k+1];
            if (bmax < b[i-k+1]) bmax = b[i-k+1];

```



```

    if (s[i] > s[i-k] + lsum * bmax){
        s[i] = s[i-k] + lsum * bmax;
        kay[i] = k; }
    }
    s[i] += header;
}
}

```

### 15.2.3 矩阵乘法链

$m \times n$  矩阵  $A$  与  $n \times p$  矩阵  $B$  相乘需耗费  $\Theta(mnp)$  的时间 (见第2章练习16)。我们把  $mnp$  作为两个矩阵相乘所需时间的测量值。现假定要计算三个矩阵  $A$ 、 $B$  和  $C$  的乘积, 有两种方式计算此乘积。在第一种方式中, 先用  $A$  乘以  $B$  得到矩阵  $D$ , 然后  $D$  乘以  $C$  得到最终结果, 这种乘法的顺序可写为  $(A*B)*C$ 。第二种方式写为  $A*(B*C)$ , 道理同上。尽管这两种不同的计算顺序所得的结果相同, 但时间消耗会有很大的差距。

例15-12 假定  $A$  为  $100 \times 1$  矩阵,  $B$  为  $1 \times 100$  矩阵,  $C$  为  $100 \times 1$  矩阵, 则  $A*B$  的时间耗费为 10 000, 得到的结果  $D$  为  $100 \times 100$  矩阵, 再与  $C$  相乘所需的时间耗费为 1 000 000, 因此计算  $(A*B)*C$  的总时间为 1 010 000。 $B*C$  的时间耗费为 10 000, 得到的中间矩阵为  $1 \times 1$  矩阵, 再与  $A$  相乘的时间消耗为 100, 因而计算  $A*(B*C)$  的时间耗费竟只有 10 100! 而且, 计算  $(A*B)*C$  时, 还需 10 000 个单元来存储  $A*B$ , 而  $A*(B*C)$  计算过程中, 只需用 1 个单元来存储  $B*C$ 。

下面举一个得益于选择合适秩序计算  $A*B*C$  矩阵的实例: 考虑两个 3 维图像的匹配。图像匹配问题的要求是, 确定一个图像需旋转、平移和缩放多少次才能逼近另一个图像。实现匹配的方法之一便是执行约 100 次迭代计算, 每次迭代需计算  $12 \times 1$  个向量  $T$ :

$$T = A(x, y, z) * B(x, y, z) * C(x, y, z)$$

其中  $A$ ,  $B$  和  $C$  分别为  $12 \times 3$ ,  $3 \times 3$  和  $3 \times 1$  矩阵。 $(x, y, z)$  为矩阵中向量的坐标。设  $t$  表示计算  $A(x, y, z) * B(x, y, z) * C(x, y, z)$  的计算量。假定此图像含  $256 \times 256 \times 256$  个向量, 在此条件中, 这 100 个迭代所需的总计算量近似为  $100 * 256^3 * t = 1.7 * 10^9 t$ 。若三个矩阵是按由左向右的顺序相乘的, 则  $t = 12 * 3 * 3 + 12 * 3 * 1 = 144$ ; 但如果从右向左相乘,  $t = 3 * 3 * 1 + 12 * 3 * 1 = 45$ 。由左至右计算约需  $2.4 * 10^{11}$  个操作, 而由右至左计算大概只需  $7.5 * 10^{10}$  个操作。假如使用一个每秒可执行 1 亿次操作的计算机, 由左至右需 40 分钟, 而由右至左只需 12.5 分钟。

在计算矩阵运算  $A*B*C$  时, 仅有两种乘法顺序 (由左至右或由右至左), 所以可以很容易算出每种顺序所需要的操作数, 并选择操作数比较少的那种乘法顺序。但对于更多矩阵相乘来说, 情况要复杂得多。如计算矩阵乘积  $M_1 \times M_2 \times \dots \times M_q$ , 其中  $M_i$  是一个  $r_i \times r_{i+1}$  矩阵 ( $1 \leq i \leq q$ )。不妨考虑  $q=4$  的情况, 此时矩阵运算  $A*B*C*D$  可按以下方式 (顺序) 计算:

$$A*((B*C)*D) \quad A*(B*(C*D)) \quad (A*B)*(C*D) \quad (A*(B*C))*D$$

不难看出计算的方法数会随  $q$  以指数级增加。因此, 对于很大的  $q$  来说, 考虑每一种计算顺序并选择最优者已是不切实际的。

现在要介绍一种采用动态规划方法获得矩阵乘法次序的最优策略。这种方法可将算法的时间消耗降为  $\Theta(q^3)$ 。用  $M_{ij}$  表示链  $M_i \times \dots \times M_j$  ( $i \leq j$ ) 的乘积。设  $c(i, j)$  为用最优法计算  $M_{ij}$  的消耗,  $kay(i, j)$  为用最优法计算  $M_{ij}$  的最后一步  $M_{ik} \times M_{k+1, j}$  的消耗。因此  $M_{ij}$  的最优算法包括如何

用最优算法计算  $M_{ik}$  和  $M_{kj}$  以及计算  $M_{ik} \times M_{kj}$ 。根据最优原理，可得到如下的动态规划递归式：

$$c(i, i) = 0, 1 \leq i \leq q$$

$$c(i, i+1) = r_i r_{i+1} r_{i+2}; \text{ kay}(i, i+1) = i, 1 \leq i < q$$

$$c(i, i+s) = \min_{i \leq k < i+s} \{c(i, k) + c(k+1, i+s) + r_i r_{k+1} r_{i+s+1}\};$$

$$\text{kay}(i, i+s) = \text{获得上述最小值的 } k$$

$$1 \leq i \leq q-s, 1 < s < q$$

以上求  $c$  的递归式可用递归或迭代的方法来求解。 $c(1, q)$  为用最优法计算矩阵链的消耗， $\text{kay}(1, q)$  为最后一步的消耗。其余的乘积可由  $\text{kay}$  值来确定。

### 1. 递归方法

与求解0/1背包及图像压缩问题一样，本递归方法也须避免重复计算  $c(i, j)$  和  $\text{kay}(i, j)$ ，否则算法的复杂性将会非常高。

例15-13 设  $q=5$  和  $r = (10, 5, 1, 10, 2, 10)$ ，由动态规划的递归式得：

$$c(1, 5) = \min\{c(1, 1) + c(2, 5) + 500, c(1, 2) + c(3, 5) + 100, \\ c(1, 3) + c(4, 5) + 1000, c(1, 4) + c(5, 5) + 200\} \quad (15-4)$$

式中待求的  $c$  中有四个  $c$  的  $s=0$  或  $1$ ，因此用动态规划方法可立即求得它们的值： $c(1, 1)=c(5, 5)=0$ ； $c(1, 2)=50$ ； $c(4, 5)=200$ 。现计算  $C(2, 5)$ ：

$$c(2, 5) = \min\{c(2, 2) + c(3, 5) + 50, c(2, 3) + c(4, 5) + 500, c(2, 4) + c(5, 5) + 100\} \quad (15-5)$$

其中  $c(2, 2)=c(5, 5)=0$ ； $c(2, 3)=50$ ； $c(4, 5)=200$ 。再用递归式计算  $c(3, 5)$  及  $c(2, 4)$ ：

$$c(3, 5) = \min\{c(3, 3) + c(4, 5) + 100, c(3, 4) + c(5, 5) + 20\} = \min\{0 + 200 + 100, 20 + 0 + 20\} = 40$$

$$c(2, 4) = \min\{c(2, 2) + c(3, 4) + 10, c(2, 3) + c(4, 4) + 100\} = \min\{0 + 20 + 10, 50 + 10 + 20\} = 30$$

由以上计算还可得  $\text{kay}(3, 5)=4$ ， $\text{kay}(2, 4)=2$ 。现在，计算  $c(2, 5)$  所需的所有中间值都已求得，将它们代入式 (15-5) 得：

$$c(2, 5) = \min\{0 + 40 + 50, 50 + 200 + 500, 30 + 0 + 100\} = 90 \text{ 且 } \text{kay}(2, 5)=2$$

再用式 (15-4) 计算  $c(1, 5)$ ，在此之前必须算出  $c(3, 5)$ 、 $c(1, 3)$  和  $c(1, 4)$ 。同上述过程，亦可计算出它们的值分别为40、150和90，相应的  $\text{kay}$  值分别为4、2和2。代入式 (15-4) 得：

$$c(1, 5) = \min\{0 + 90 + 500, 50 + 40 + 100, 150 + 200 + 1000, 90 + 0 + 200\} = 190 \text{ 且 } \text{kay}(1, 5)=2$$

此最优乘法算法的消耗为 190，由  $\text{kay}(1, 5)$  值可推出该算法的最后一步， $\text{kay}(1, 5)$  等于 2，因此最后一步为  $M_{12} \times M_{35}$ ，而  $M_{12}$  和  $M_{35}$  都是用最优法计算而来。由  $\text{kay}(1, 2)=1$  知  $M_{12}$  等于  $M_{11} \times M_{22}$ ，同理由  $\text{kay}(3, 5)=4$  得知  $M_{35}$  由  $M_{34} \times M_{55}$  算出。依此类推， $M_{34}$  由  $M_{33} \times M_{44}$  得出。因而此最优乘法算法的步骤为：

$$M_{11} \times M_{22} = M_{12}$$

$$M_{33} \times M_{44} = M_{34}$$

$$M_{34} \times M_{55} = M_{35}$$

$$M_{12} \times M_{35} = M_{15}$$

计算  $c(i, j)$  和  $\text{kay}(i, j)$  的递归代码见程序 15-6。在函数  $C$  中， $r$  为全局一维数组变量， $\text{kay}$  是全局二维数组变量，函数  $C$  返回  $c(i, j)$  之值且置  $\text{kay}[a][b]=\text{kay}(a, b)$  (对于任何  $a, b$ )，其中  $c(a, b)$  在计算  $c(i, j)$  时皆已算出。函数 Traceback 利用函数  $C$  中已算出的  $\text{kay}$  值来推导出最优乘法算法

的步骤。

设  $t(q)$  为函数  $C$  的复杂性, 其中  $q=j-i+1$  (即  $M_{ij}$  是  $q$  个矩阵运算的结果)。当  $q$  为 1 或 2 时,  $t(q)=d$ , 其中  $d$  为一常数; 而  $q>2$  时,  $t(q)=2 \sum_{k=1}^{q-1} t(k)+eq$ , 其中  $e$  是一个常量。因此当  $q>2$  时,  $t(q)>2t(q-1)+e$ , 所以  $t(q)=O(2^q)$ 。函数 Traceback 的复杂性为  $O(q)$ 。

程序 15-6 递归计算  $c(i, j)$  和  $kay(i, j)$

```
int C(int i, int j)
{ // 返回  $c(i, j)$  且计算  $k(i, j) = kay[i][j]$ 
  if (i == j) return 0; // 一个矩阵的情形
  if (i == j-1) { // 两个矩阵的情形
    kay[i][i+1] = i;
    return r[i]*r[i+1]*r[i+2];
  } // 多于两个矩阵的情形
  // 设  $u$  为  $k=i$  时的最小值
  int u = C(i, i) + C(i+1, j) + r[i]*r[i+1]*r[j+1];
  kay[i][j] = i;

  // 计算其余的最小值并更新  $u$ 
  for (int k = i+1; k < j; k++) {
    int t = C(i, k) + C(k+1, j) + r[i]*r[k+1]*r[j+1];
    if (t < u) { // 小于最小值的情形
      u = t;
      kay[i][j] = k;
    }
  }
  return u;
}

void Traceback (int i, int j, int **kay)
{ // 输出计算  $M_{ij}$  的最优方法
  if (i == j) return;
  Traceback(i, kay[i][j], kay);
  Traceback(kay[i][j]+1, j, kay);
  cout << "Multiply M" << i << ", " << kay[i][j];
  cout << " and M " << (kay[i][j]+1) << ", " << j << endl;
}
```

## 2. 无重复计算的递归方法

若避免再次计算前面已经计算过的  $c$  (及相应的  $kay$ ), 可将复杂性降低到  $O(q^3)$ 。而为了避免重复计算, 需用一个全局数组  $c[ ][ ]$  存储  $c(i, j)$  值, 该数组初始值为 0。函数  $C$  的新代码见程序 15-7:

程序 15-7 无重复计算的  $c(i, j)$  计算方法

```
int C(int i, int j)
{ // 返回  $c(i, j)$  并计算  $kay(i, j) = kay[i][j]$ 
  // 避免重复计算

  // 检查是否已计算过
  if (c[i][j] > 0) return c[i][j];
```

```

//若未计算,则进行计算
if(i==j) return 0; //一个矩阵的情形
if(i==j-1){//两个矩阵的情形
    kay[i][i+1]=i;
    c[i][j]=r[i]*r[i+1]*r[i+2];
    return c[i][j];}
//多于两个矩阵的情形
//设u为k = i时的最小值
int u=C(i,i)+C(i+1,j)+r[i]*r[i+1]*r[j+1];
kay[i][j]=i;

//计算其余的最小值并更新u
for (int k=i+1; k<j;k++){
    int t=C(i,k)+C(k+1,j)+r[i]*r[k+1]*r[j+1];
    if (t<u) {// 比最小值还小
        u=t;
        kay[i][j]=k;}
}
c[i][j]=u;
return u;
}

```

为分析改进后函数C的复杂性,再次使用分期计算方法。注意到调用C(1, q)时每个 $c(i, j)$  ( $1 \leq i < j \leq q$ )仅被计算一次。要计算尚未计算过的 $c(a, b)$ ,需附加的工作量 $s=j-i > 1$ 。将s计入第一次计算 $c(a, b)$ 时的工作量中。在依次计算 $c(a, b)$ 时,这个s会转计到每个 $c(a, b)$ 的第一次计算时间c中,因此每个 $c(i, i)$ 均被计入 $s$ 。对于每个s,有 $q-s+1$ 个 $c(i, j)$ 需要计算,因此总的工作消耗为 $\sum_{s=1}^{q-1} (q-s+1) = \Theta(q^3)$ 。

### 3. 迭代方法

c的动态规划递归式可用迭代的方法来求解。若按 $s=2, 3, \dots, q-1$ 的顺序计算 $c(i, i+s)$ ,每个c和kay仅需计算一次。

**例15-14** 考察例15-13中五个矩阵的情况。先初始化 $c(i, i)$  ( $0 \leq i \leq 5$ )为0,然后对于 $i=1, \dots, 4$ 分别计算 $c(i, i+1)$ 。 $c(1, 2)=r_1 r_2 r_3=50$ ,  $c(2, 3)=50$ ,  $c(3, 4)=20$ 和 $c(4, 5)=200$ 。相应的kay值分别为1, 2, 3和4。

当 $s=2$ 时,可得:

$$c(1,3)=\min\{c(1,1)+c(2,3)+r_1 r_2 r_4, c(1,2)+c(3,3)+r_1 r_3 r_4\}=\min\{0+50+500, 50+0+100\}=150$$

且 $kay(1,3)=2$ 。用相同方法可求得 $c(2,4)$ 和 $c(3,5)$ 分别为30和40,相应kay值分别为2和3。

当 $s=3$ 时,需计算 $c(1,4)$ 和 $c(2,5)$ 。计算 $c(2,5)$ 所需要的所有中间值均已知(见(15-5)式),代入计算公式后可得 $c(2,5)=90$ ,  $kay(2,5)=2$ 。 $c(1,4)$ 可用同样的公式计算。最后,当 $s=4$ 时,可直接用(15-4)式来计算 $c(1,5)$ ,因为该式右边所有项都已知。

计算c和kay的迭代程序见函数MatrixChain(见程序15-8),该函数的复杂性为 $\Theta(q^3)$ 。计算出kay后同样可用程序15-6中的Traceback函数推算出相应的最优乘法计算过程。

程序15-8 c和kay的迭代计算

```
void MatrixChain(int r[], int q, int **c, int **kay)
```

```

// 为所有的Mij 计算耗费和 kay
// 初始化c[i][i], c[i][i+1]和 kay[i][i+1]

for (int i = 1; i < q; i++) {
    c[i][i] = 0;
    c[i][i+1] = r[i]*r[i+1]*r[i+2];
    kay[i][i+1] = i;
}
c[q][q] = 0;

//计算余下的 c和kay
for (int s = 2; s < q; s++)
for (int i = 1; i <= q - s; i++) {
    // k = i时的最小项
    c[i][i+s] = c[i][i] + c[i+1][i+s] + r[i]*r[i+1]*r[i+s+1];
    kay[i][i+s] = i;

    // 余下的最小项
    for (int k = i+1; k < i + s; k++) {
        int t = c[i][k] + c[k+1][i+s] + r[i]*r[k+1]*r[i+s+1];
        if (t < c[i][i+s]) { // 更小的最小项
            c[i][i+s] = t;
            kay[i][i+s] = k;
        }
    }
}
}

```

### 15.2.4 最短路径

假设 $G$ 为有向图，其中每条边都有一个长度（或耗费），图中每条有向路径的长度等于该路径中各边的长度之和。对于每对顶点 $(i, j)$ ，在顶点 $i$ 与 $j$ 之间可能有多条路径，各路径的长度可能各不相同。我们定义从 $i$ 到 $j$ 的所有路径中，具有最小长度的路径为从 $i$ 到 $j$ 的最短路径。

例15-15 如图15-4所示。从顶点1到顶点3的路径有

- 1) 1,2,5,3
- 2) 1,4,3
- 3) 1,2,5,8,6,3
- 4) 1,4,6,3

由该图可知,各路径相应的长度为10、28、9、27，因而路径3) 是该图中顶点1到顶点3的最短路径。

在所有点对最短路径问题（all-pairs shortest-paths problem）中，要寻找有向图 $G$ 中每对顶点之间的最短路径。也就是说，对于每对顶点 $(i, j)$ ，需要寻找从 $i$ 到 $j$ 的最短路径及从 $j$ 到 $i$ 的最短路径。因此对于一个 $n$ 个顶点的图来说，需寻找 $p=n(n-1)$ 条最短路径。假定图 $G$ 中不

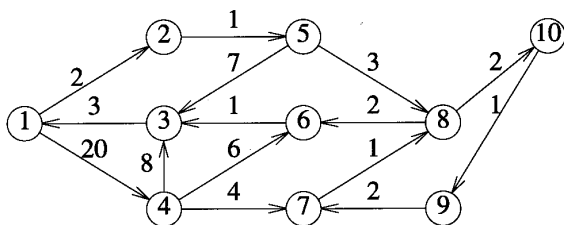


图15-4 有向图

含有长度为负数的环路，只有在这种假设下才可保证  $G$  中每对顶点  $(i, j)$  之间总有一条不含环路的最短路径。当有向图中存在长度小于 0 的环路时，可能得到长度为  $-\infty$  的更短路径，因为包含该环路的最短路径往往可无限多次地加上此负长度的环路。

设图  $G$  中  $n$  个顶点的编号为 1 到  $n$ 。令  $c(i, j, k)$  表示从  $i$  到  $j$  的最短路径的长度，其中  $k$  表示该路径中的最大顶点。因此，如果  $G$  中包含边  $\langle i, j \rangle$ ，则  $c(i, j, 0) = \text{边} \langle i, j \rangle$  的长度；若  $i=j$ ，则  $c(i, j, 0)=0$ ；如果  $G$  中不包含边  $\langle i, j \rangle$ ，则  $c(i, j, 0)=+\infty$ 。 $c(i, j, n)$  则是从  $i$  到  $j$  的最短路径的长度。

例15-6 考察图15-4。若  $k=0, 1, 2, 3$ ，则  $c(1, 3, k)=\infty$ ； $c(1, 3, 4)=28$ ；若  $k=5, 6, 7$ ，则  $c(1, 3, k)=10$ ；若  $k=8, 9, 10$ ，则  $c(1, 3, k)=9$ 。因此1到3的最短路径长度为9。

对于任意  $k > 0$ ，如何确定  $c(i, j, k)$  呢？中间顶点不超过  $k$  的  $i$  到  $j$  的最短路径有两种可能：该路径含或不含中间顶点  $k$ 。若不含，则该路径长度应为  $c(i, j, k-1)$ ，否则长度为  $c(i, k, k-1) + c(k, j, k-1)$ 。 $c(i, j, k)$  可取两者中的最小值。因此可得到如下递归式：

$$c(i, j, k) = \min\{c(i, j, k-1), c(i, k, k-1) + c(k, j, k-1)\}, k > 0$$

以上的递归公式将一个  $k$  级运算转化为多个  $k-1$  级运算，而多个  $k-1$  级运算应比一个  $k$  级运算简单。如果用递归方法求解上式，则计算最终结果的复杂性将无法估量。令  $t(k)$  为递归求解  $c(i, j, k)$  的时间。根据递归式可以看出  $t(k)=2t(k-1)+c$ 。利用替代方法可得  $t(n)=\Theta(2^n)$ 。因此得到所有  $c(i, j, n)$  的时间为  $\Theta(n^2 2^n)$ 。

当注意到某些  $c(i, j, k-1)$  值可能被使用多次时，可以更高效地求解  $c(i, j, n)$ 。利用避免重复计算  $c(i, j, k)$  的方法，可将计算  $c$  值的时间减少到  $\Theta(n^3)$ 。这可通过递归方式（见程序15-7矩阵链问题）或迭代方式来实现。出迭代算法的伪代码如图15-5所示。

```
//寻找最短路径的长度
//初始化  $c(i, j, 1)$ 
for (int i=1; i<=n; i++)
    for (int j=1; j<=n; j++)
         $c(i, j, 0) = a(i, j)$ ; //  $a$  是长度邻接矩阵
//计算  $c(i, j, k) (0 < k <= n)$ 
for (int k=1; k<=n; k++)
    for (int i=1; i<=n; i++)
        for (int j=1; j<=n; j++)
            if ( $c(i, k, k-1) + c(k, j, k-1) < c(i, j, k-1)$ )
                 $c(i, j, k) = c(i, k, k-1) + c(k, j, k-1)$ ;
            else  $c(i, j, k) = c(i, j, k-1)$ ;
```

图15-5 最短路径算法的伪代码

注意到对于任意  $i, j$ ， $c(i, k, k)=c(i, k, k-1)$  且  $c(k, i, k)=c(k, i, k-1)$ ，因而，若用  $c(i, j)$  代替图15-5的  $c(i, j, k)$ ，最后所得的  $c(i, j)$  之值将等于  $c(i, j, n)$  值。此时图15-5可改写成程序15-9的C++代码。程序15-9中还利用了程序12-1中定义的AdjacencyWDigraph类。函数AllPairs在c中返回最短路径的长度。若  $i$  到  $j$  无通路，则  $c[i][j]$  被赋值为NoEdge。函数AllPairs同时计算了  $kay[i][j]$ ，其中  $kay[i][j]$  表示从  $i$  到  $j$  的最短路径中最大的  $k$  值。在后面将看到如何根据  $kay$  值来推断出从一个顶点到另一顶点的最短路径（见程序15-10中的函数OutputPath）。

程序15-9的时间复杂性为  $\Theta(n^3)$ ，其中输出一条最短路径的实际时间为  $O(n)$ 。

程序15-9 c 和kay 的计算

---

```
template<class T>
void AdjacencyWDigraph<T>::Allpairs(T **c, int **kay)
{
    //所有点对的最短路径
    //对于所有 i和j，计算c[i][j]和kay[i][j]
    //初始化c[i][j]=c(i, j, 0)
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++) {
            c[i][j] = a[i][j];
            kay[i][j] = 0;
        }
    for (i = 1; i <= n; i++)
        c[i][i] = 0;

    // 计算c[i][j] = c(i,j,k)
    for (int k = 1; k <= n; k++)
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++) {
                T t1 = c[i][k];
                T t2 = c[k][j];
                T t3 = c[i][j];
                if (t1 != NoEdge && t2 != NoEdge && (t3 == NoEdge || t1 + t2 < t3)) {
                    c[i][j] = t1 + t2;
                    kay[i][j] = k;
                }
            }
}
```

---

程序15-10 输出最短路径

---

```
void outputPath(int **kay, int i, int j)
{
    // 输出 i 到 j 的路径的实际代码
    if (i == j) return;
    if (kay[i][j] == 0) cout << j << ' ';
    else {outputPath(kay, i, kay[i][j]);
        outputPath(kay, kay[i][j], j);}
}

template<class T>
void OutputPath(T **c, int **kay, T NoEdge, int i, int j)
{
    // 输出从 i 到 j 的最短路径
    if (c[i][j] == NoEdge) {
        cout << "There is no path from " << i << " to " << j << endl;
        return;
    }
    cout << "The path is" << endl;
    cout << i << ' ';
    outputPath(kay, i, j);
}
```



```
cout << endl;
}
```

例15-17 图15-6a 给出某图的长度矩阵a, 15-6b 给出由程序15-9所计算出的c 矩阵, 15-6c 为对应的kay值。根据15-6c 中的kay 值, 可知从1到5的最短路径是从1到kay[1][5]=4的最短路径再加上从4到5的最短路径, 因为kay[4][5]=0, 所以从4到5的最短路径无中间顶点。从1到4的最短路径经过kay[1][4]=3。重复以上过程, 最后可得1到5的最短路径为: 1, 2, 3, 4, 5。

0 1 4 4 8

3 0 1 5 9

2 2 0 1 8

8 8 9 0 1

8 8 2 9 0

a)

0 1 2 3 4

3 0 1 2 3

2 2 0 1 2

5 5 3 0 1

4 4 2 3 0

b)

0 0 2 3 4

0 0 0 3 4

0 0 0 0 4

5 5 5 0 0

3 3 0 3 0

c)

图15-6 最短路径的例子

### 15.2.5 网络的无交叉子集

在11.5.3节的交叉分布问题中, 给定一个每边带  $n$  个针脚的布线通道和一个排列  $C$ 。顶部的针脚  $i$  与底部的针脚  $C_i$  相连, 其中  $1 \leq i \leq n$ , 数对  $(i, C_i)$  称为网组。总共有  $n$  个网组需连接或连通。假设有两个或更多的布线层, 其中有一个为优先层, 在优先层中可以使用更细的连线, 其电阻也可能比其他层要小得多。布线时应尽可能在优先层中布设更多的网组。而剩下的其他网组将布设在其他层。当且仅当两个网组之间不交叉时, 它们可布设在同一层。我们的任务是寻找一个最大无交叉子集 (Maximum Noncrossing Subset, MNS)。在该集中, 任意两个网组都不交叉。因  $(i, C_i)$  完全由  $i$  决定, 因此可用  $i$  来指定  $(i, C_i)$ 。

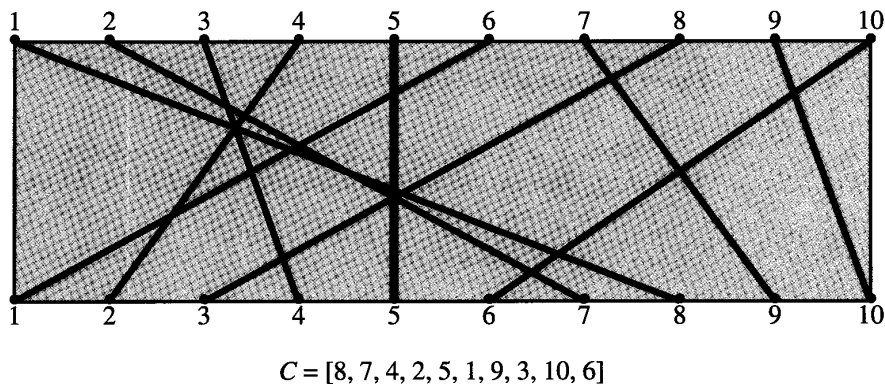


图15-7 布线举例

例15-18 考察图15-7 (对应于图10-17)。(1,8)和(2,7) (也即1号网组和2号网组) 交叉, 因而不能布设在同一层中。而(1,8), (7,9) 和(9,10) 未交叉, 因此可布设在同一层。但这3个网组并

不能构成一个 MNS，因为还有更大的不交叉子集。图 10-17 中给出的例子中，集合  $\{(4,2), (5,5), (7,9), (9,10)\}$  是一个含 4 个网组的 MNS。

设  $MNS(i, j)$  代表一个 MNS，其中所有的  $(u, C_u)$  满足  $u \leq i, C_u \leq j$ 。令  $size(i, j)$  表示  $MNS(i, j)$  的大小(即网组的数目)。显然  $MNS(n, n)$  是对应于给定输入的 MNS，而  $size(n, n)$  是它的大小。

例 15-19 对于图 10-17 中的例子， $MNS(10, 10)$  是我们要找的最终结果。如例 15-18 中所指出的， $size(10, 10) = 4$ ，因为  $(1, 8), (2, 7), (7, 9), (8, 3), (9, 10)$  和  $(10, 6)$  中要么顶部针脚编号比 7 大，要么底部针脚编号比 6 大，因此它们都不属于  $MNS(7, 6)$ 。因此只需考察剩下的 4 个网组是否属于  $MNS(7, 6)$ ，如图 15-8 所示。子集  $\{(3, 4), (5, 5)\}$  是大小为 2 的无交叉子集。没有大小为 3 的无交叉子集，因此  $size(7, 6) = 2$ 。

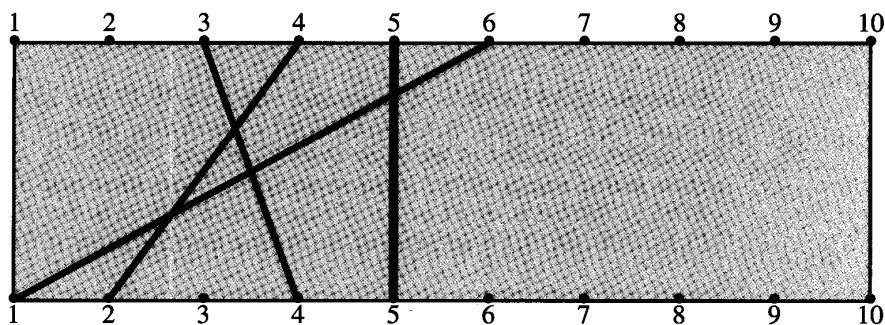


图 15-8 图 15-7 中可能属于  $MNS(7, 6)$  的网组

当  $i=1$  时， $(1, C_1)$  是  $MNS(1, j)$  的唯一候选。仅当  $j \geq C_1$  时，这个网组才会是  $MNS(1, j)$  的一个成员，即：

$$size(i, j) = \begin{cases} 0 & \text{if } j < C_1 \\ 1 & \text{if } j \geq C_1 \end{cases} \quad (15-6)$$

下一步，考虑  $i > 1$  时的情况。若  $j < C_i$ ，则  $(i, C_i)$  不可能是  $MNS(i, j)$  的成员，所有属于  $MNS(i, j)$  的  $(u, C_u)$  都需满足  $u < i$  且  $C_u < j$ ，因此：

$$size(i, j) = size(i-1, j), \quad j < C_i \quad (15-7)$$

若  $j \geq C_i$ ，则  $(i, C_i)$  可能在也可能不在  $MNS(i, j)$  内。若  $(i, C_i)$  在  $MNS(i, j)$  内，则在  $MNS(i, j)$  中不会有这样的  $(u, C_u)$ ： $u < i$  且  $C_u > C_i$ ，因为这个网组必与  $(i, C_i)$  相交。因此  $MNS(i, j)$  中的其他所有成员都必须满足条件  $u < i$  且  $C_u \leq C_i$ 。在  $MNS(i, j)$  中这样的网组共有  $M_{i-1, C_i}$  个。若  $(i, C_i)$  不在  $MNS(i, j)$  中，则  $MNS(i, j)$  中的所有  $(u, C_u)$  必须满足  $u < i$ ；因此  $size(i, j) = size(i-1, j)$ 。虽然不能确定  $(i, C_i)$  是否在  $MNS(i, j)$  中，但我们可以根据获取更大 MNS 的原则来作出选择。因此：

$$size(i, j) = \max\{size(i-1, j), size(i-1, C_i-1)+1\}, \quad j \geq C_i \quad (15-8)$$

虽然从 (15-6) 式到 (15-8) 式可用递归法求解，但从前面的例子可以看出，即使避免了重复计算，动态规划递归算法的效率也不够高，因此只考虑迭代方法。在迭代过程中先用式 (15-6) 计算出  $size(1, j)$ ，然后再用式 (15-7) 和 (15-8) 按  $i=2, 3, \dots, n$  的顺序计算  $size(i, j)$ ，最后再用 Traceback 来得到  $MNS(n, n)$  中的所有网组。

例 15-20 图 15-9 给出了图 15-7 对应的  $size(i, j)$  值。因  $size(10, 10) = 4$ ，可知 MNS 含 4 个网组。为求

得这4个网组，先从 $size(10,10)$ 入手。可用(15-8)式算出 $size(10,10)$ 。根据式(15-8)时的产生原因可知 $size(10,10)=size(9,10)$ ，因此现在要求 $MNS(9,10)$ 。由于 $MNS(10,10) \supset size(8,10)$ ，因此 $MNS(9,10)$ 中必包含9号网组。 $MNS(9,10)$ 中剩下的网组组成 $MNS(8, C_9-1)=MNS(8,9)$ 。由 $MNS(8,9)=MNS(7,9)$ 知，8号网组可以被排除。接下来要求 $MNS(7,9)$ ，因为 $size(7,9) \supset size(6,9)$ ，所以 $MNS$ 中必含7号网组。 $MNS(7,9)$ 中余下的网组组成 $MNS(6, C_7-1)=MNS(6,8)$ 。根据 $size(6,8)=size(5,8)$ 可排除6号网组。按同样的方法，5号网组，3号网组加入 $MNS$ 中，而4号网组等其他网组被排除。因此回溯过程所得到的大小为4的 $MNS$ 为 $\{3,5,7,9\}$ 。

$i$	$j$									
	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	1	1	1
2	0	0	0	0	0	0	1	1	1	1
3	0	0	0	1	1	1	1	1	1	1
4	0	1	1	1	1	1	1	1	1	1
5	0	1	1	1	2	2	2	2	2	2
6	1	1	1	1	2	2	2	2	2	2
7	1	1	1	1	2	2	2	2	3	3
8	1	1	2	2	2	2	2	2	3	3
9	1	1	2	2	2	2	2	2	3	4
10	1	1	2	2	2	3	3	3	3	4

图15-9 图15-7对应的 $size(i, j)$

注意到在回溯过程中未用到 $size(10, j)$  ( $j < 10$ )，因此不必计算这些值。

程序15-11给出了计算 $size(i, j)$ 的迭代代码和输出 $MNS$ 的代码。函数 $MNS$ 用来计算 $size(i, j)$ 的值，计算结果用一个二维数组 $MN$ 来存储。 $size[i][j]$ 表示 $size(i, j)$ ，其中 $i=j=n$ 或 $1 \leq i < n$ ， $0 \leq j < n$ ，计算过程的时间复杂性为 $\Theta(n^2)$ 。函数 $Traceback$ 在 $Net[0:m-1]$ 中输出所得到的 $MNS$ ，其时间复杂性为 $\Theta(n)$ 。因此求解MMS问题的动态规划算法的总的时间复杂性为 $\Theta(n^2)$ 。

程序15-11 寻找最大无交叉子集

```
void MNS(int C[], int n, int **size)
//对于所有的 i 和 j，计算size[i][j]

//初始化size[1][*]
for (int j = 0; j < C[1]; j++)
    size[1][j] = 0;
for (j = C[1]; j <= n; j++)
    size[1][j] = 1;

// 计算size[i][*], 1 < i < n
for (int i = 2; i < n; i++) {
    for (int j = 0; j < C[i]; j++)
        size[i][j] = size[i-1][j];
    for (j = C[i]; j <= n; j++)
```

```

    size[i][j] = max(size[i-1][j], size[i-1][C[i]-1]+1);
}

size[n][n] = max(size[n-1][n], size[n-1][C[n]-1]+1);
}

void Traceback(int C[], int **size, int n, int Net[], int& m)
{// 在 Net[0:m-1]中返回MMS
    int j = n; // 所允许的底部最大引脚编号
    m = 0;     // 网组的游标
    for (int i = n; i > 1; i--)
        // i 号net在 MNS中?
        if (size[i][j] != size[i-1][j]){// 在MNS中
            Net[m++] = i;
            j = C[i] - 1;}

    // 1号网组在 MNS中?
    if (j >= C[1])
        Net[m++] = 1; // 在MNS中
}

```

### 15.2.6 元件折叠

在设计电路的过程中，工程师们会采取多种不同的设计风格。其中的两种为位 - 片设计 (bit-slice design) 和标准单元设计 (standard-cell design)。在前一种方法中，电路首先被设计为一个元件栈 (如图 15-10a 所示)。每个元件  $C_i$  宽为  $w_i$ ，高为  $h_i$ ，而元件宽度用片数来表示。图

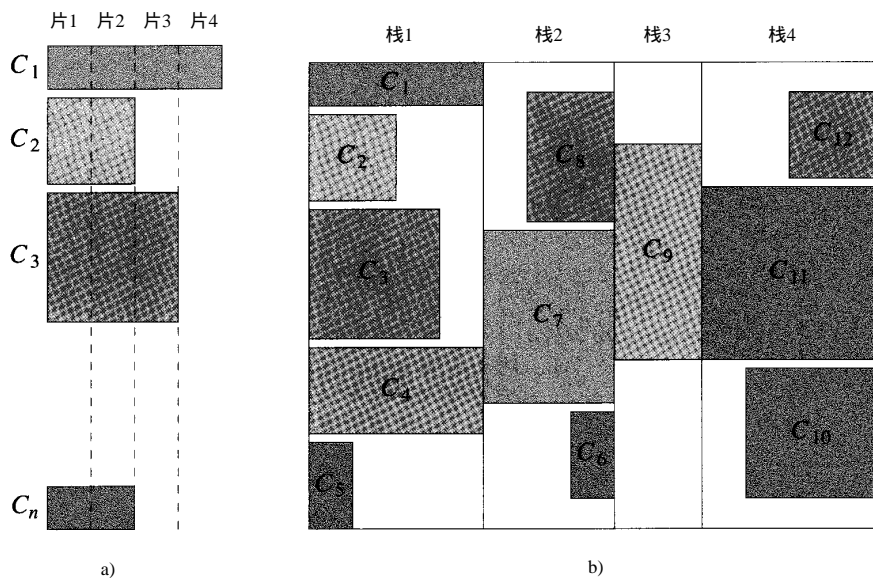


图15-10 元件栈及其折叠

a) 四片栈 b) 折叠



15-10a 给出了一个四片的设计。线路是按片来连接各元件的，即连线可能连接元件  $C_i$  的第  $j$  片到元件  $C_{i+1}$  的第  $j$  片。如果某些元件的宽度不足  $j$  片，则这些元件之间不存在片  $j$  的连线。当图 15-10a 的位 - 片设计作为某一大系统的一部分时，则在 VLSI (Very Large Scale Integrated) 芯片上为它分配一定数量的空间单元。分配是按空间宽度或高度的限制来完成的。现在的问题便是如何将元件栈折叠到分配空间中去，以便尽量减小未受限制的尺度（如，若高度限制为  $H$  时，必须折叠栈以尽量减小宽度  $W$ ）。由于其他尺度不变，因此缩小一个尺度（如  $W$ ）等价于缩小面积。

可用折线方式来折叠元件栈，在每一折叠点，元件旋转  $180^\circ$ 。在图 15-10b 的例子中，一个 12 元件的栈折叠成四个垂直栈，折叠点为  $C_6$ ,  $C_9$  和  $C_{10}$ 。折叠栈的宽度是宽度最大的元件所需的片数。在图 15-10b 中，栈宽各为 4, 3, 2 和 4。折叠栈的高度等于各栈所有元件高度之和的最大值。在图 15-10b 中栈 1 的元件高度之和最大，该栈的高度决定了包围所有栈的矩形高度。

实际上，在元件折叠问题中，还需考虑连接两个栈的线路所需的附加空间。如，在图 15-10b 中  $C_5$  和  $C_6$  间的线路因  $C_6$  为折叠点而弯曲。这些线路要求在  $C_5$  和  $C_6$  之下留有垂直空间，以便能从栈 1 连到栈 2。令  $r_i$  为  $C_i$  是折叠点时所需的高度。栈 1 所需的高度为  $\sum_{i=1}^5 h_i + r_6$ ，栈 2 所需高度为  $\sum_{i=6}^8 h_i + r_9 + r_{10}$ 。

在标准单元设计中，电路首先被设计成为具有相同高度的符合线性顺序的元件排列。假设此线性顺序中的元件为  $C_1, \dots, C_n$ ，下一步元件被折叠成如图 15-11 所示的相同宽度的行。在此图中，12 个标准单元折叠成四个等宽行。折叠点是  $C_4$ ,  $C_6$  和  $C_{11}$ 。在相邻标准单元行之间，使用布线通道来连接不同的行。折叠点决定了所需布线通道的高度。设  $l_i$  表示当  $C_i$  为折叠点时所需的通道高度。在图 15-11 的例子中，布线通道 1 的高度为  $l_4$ ，通道 2 的高度为  $l_6$ ，通道 3 的高度为  $l_{11}$ 。

位 - 片栈折叠和标准单元折叠都会引出一系列的问题，这些问题可用动态规划方法来解决。

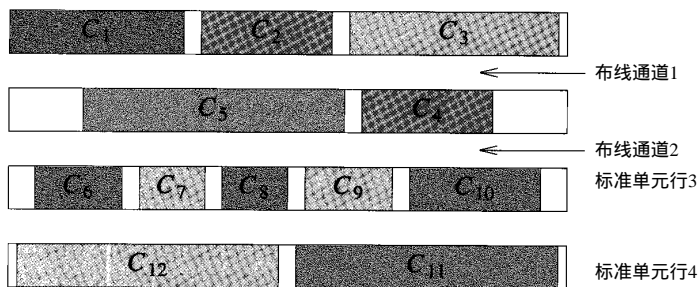


图15-11 标准单元折叠

### 1. 等宽位 - 片元件折叠

定义  $r_1 = r_{n+1} = 0$ 。由元件  $C_i$  至  $C_j$  构成的栈的高度要求为  $\sum_{k=i}^j l_k + r_i + r_{j+1}$ 。设一个位 - 片设计中所有元件有相同宽度  $W$ 。首先考察在折叠矩形的高度  $H$  给定的情况下，如何缩小其宽度。设  $W_i$  为将元件  $C_i$  到  $C_n$  折叠到高为  $H$  的矩形时的最小宽度。若折叠不能实现（如当  $r_i + h_i > H$  时），取  $W_i = \infty$ 。注意到  $W_i$  可能是所有  $n$  个元件的最佳折叠宽度。

当折叠  $C_i$  到  $C_n$  时，需要确定折叠点。现假定折叠点是按栈左到栈右的顺序来取定的。若第一点定为  $C_{k+1}$ ，则  $C_i$  到  $C_k$  在第一个栈中。为了得到最小宽度，从  $C_{k+1}$  到  $C_n$  的折叠必须用最优方法，因此又将用到最优原理，可用动态规划方法来解决此问题。当第一个折叠点  $k+1$  已知

时,可得到以下公式:

$$W_i = w + W_{k+1} \quad (15-9)$$

由于不知道第一个折叠点,因此需要尝试所有可行的折叠点,并选择满足(15-9)式的折叠点。令  $hsum(i, k) = \sum_{j=i}^k h_j$ 。因  $k+1$  是一个可行的折叠点,因此  $hsum(i, k) + r_i + r_{k+1}$  一定不会超过  $H$ 。根据上述分析,可得到以下动态规划递归式:

$$W_i = w + \min\{W_{k+1} \mid hsum(i, k) + r_i + r_{k+1} \leq H, i \leq k \leq n\} \quad (15-10)$$

这里  $W_{n+1} = 0$ , 且在无最优折叠点  $k+1$  时  $W_i$  为  $\infty$ 。利用递归式(15-10), 可通过递归计算  $W_n, W_{n-1}, \dots, W_2, W_1$  来计算  $W_i$ 。  $W_i$  的计算需要至多检查  $n-i+1$  个  $W_{k+1}$ , 耗时为  $O(n-k)$ 。因此计算所有  $W_i$  的时间为  $O(n^2)$ 。通过保留式(15-10)每次所得的  $k$  值, 可回溯地计算出各个最优的折叠点, 其时间耗费为  $O(n)$ 。

现在来考察另外一个有关等宽元件的折叠问题: 折叠后矩形的宽度  $w$  已知, 需要尽量减小其高度。因每个折叠矩形宽为  $w$ , 因此折叠后栈的最大数量为  $s = W/w$ 。令  $H_{i,j}$  为  $C_i, \dots, C_n$  折叠成一宽度为  $jw$  的矩形后的最小高度,  $H_{i,s}$  则是所有元件折叠后的最小高度。当  $j=1$  时, 不允许任何折叠, 因此:

$$H_{i,1} = hsum(i, n) + r_i, \quad 1 \leq i \leq n$$

另外, 当  $i=n$  时, 仅有一个元件, 也不可能折叠, 因此:

$$H_{n,j} = h_n + r_n, \quad 1 \leq j \leq s$$

在其他情况下, 都可以进行元件折叠。如果第一个折叠点为  $k+1$ , 则第一个栈的高度为  $hsum(i, k) + r_i + r_{k+1}$ 。其他元件必须以至多  $(j-1)*w$  的宽度折叠。为保证该折叠的最优性, 其他元件也需以最小高度进行折叠, 即:

$$H_{i,j} = \max\{hsum(i, k) + r_i + r_{k+1}, H_{k+1, j-1}\} \quad (15-11)$$

因为第一个折叠点未知, 因此必须尝试所有可能的折叠点, 然后从中找出一个使式(15-11)的右侧取最小值的点, 该点成为第一个折叠点。所得递归式为:

$$H_{i,j} = \min_{i \leq k < n} [\max\{hsum(i, k) + r_i + r_{k+1}, H_{k+1, j-1}\}] \quad (15-12)$$

可用迭代法来求解  $H_{i,j}$  ( $1 \leq i \leq n, 1 \leq j \leq s$ ), 求解的顺序为: 先计算  $j=2$  时的  $H_{i,j}$ , 再算  $j=3, \dots$ , 以此类推。对应每个  $j$  的  $H_{i,j}$  的计算时间为  $O(n^2)$ , 所以计算所有  $H_{i,j}$  的时间为  $O(sn^2)$ 。通过保存由(15-12)式计算出的每个  $k$  值, 可以采用复杂性为  $O(n)$  的回溯过程来确定各个最优的折叠点。

## 2. 变宽位 - 片元件的折叠

首先考察折叠矩形的高度  $H$  已定, 欲求最小的折叠宽度的情况。令  $W_i$  如式(15-10)所示, 按照与(15-10)式相同的推导过程, 可得:

$$W_i = \min\{wmin(i, k) + W_{k+1} \mid hsum(i, k) + r_i + r_{k+1} \leq H, i \leq k \leq n\} \quad (15-13)$$

其中  $W_{n+1} = 0$  且  $wmin(i, k) = \min_{i \leq j \leq k} \{w_j\}$ 。可用与(15-10)式一样的方法求解(15-13)式, 所需时间为  $O(n^2)$ 。

当折叠宽度  $w$  给定时, 最小高度折叠可用折半搜索方法对超过  $O(n^2)$  个可能值进行搜索来实现, 可能的高度值为  $h(i, j) + r_i + r_{j+1}$ 。在检测每个高度时, 也可用(15-13)式来确定该折叠的宽度是否小于等于  $w$ 。这种情况下总的时间消耗为  $O(n^2 \log n)$ 。

### 3. 标准单元折叠

用  $w_i$  定义单元  $C_i$  的宽度。每个单元的高度为  $h$ 。当标准单元行的宽度  $W$  固定不变时，通过减少折叠高度，可以相应地减少折叠面积。考察  $C_i$  到  $C_n$  的最小高度折叠。设第一个折叠点是  $C_{s+1}$ 。从元件  $C_{s+1}$  到  $C_n$  的折叠必须使用最小高度，否则，可使用更小的高度来折叠  $C_{s+1}$  到  $C_n$ ，从而得到更小的折叠高度。所以这里仍可使用最优原理和动态规划方法。

令  $H_{i,s}$  为  $C_i$  到  $C_n$  折叠成宽为  $W$  的矩形时的最小高度，其中第一个折叠点为  $C_{s+1}$ 。令  $wsum(i,s) = \sum_{j=i}^s w_j$ 。可假定没有宽度超过  $W$  的元件，否则不可能进行折叠。对于  $H_{n,n}$  因为只有一个元件，不存在连线问题，因此  $H_{n,n} = h$ 。对于  $H_{i,s}$  ( $1 \leq i < s \leq n$ ) 注意到如果  $wsum(i,s) > W$ ，不可能实现折叠。若  $wsum(i,s) \leq W$ ，元件  $C_i$  和  $C_{j+1}$  在相同的标准单元行中，该行下方布线通道的高度为  $l_{s+1}$  (定义  $l_{n+1} = 0$ )。因而：

$$H_{i,s} = H_{i+1,k} \quad (15-14)$$

当  $i=s < n$  时，第一个标准单元行只包含  $C_i$ 。该行的高度为  $h$  且该行下方布线通道的高度为  $l_{i+1}$ 。因  $C_{i+1}$  到  $C_n$  单元的折叠是最优的，可得：

$$H_{i,i} = \min_{i < k \leq n} \{H_{i+1,k}\} + l_{i+1} + h \quad (15-15)$$

为了寻找最小高度折叠，首先使用式 (15-14) 和 (15-15) 来确定  $H_{i,s}$  ( $1 \leq i \leq s \leq n$ )。最小高度折叠的高度为  $\min\{H_{1,s}\}$ 。可以使用回溯过程来确定最小高度折叠中的折叠点。

### 练习

1. 修改程序 15-1，使它同时计算出能导致最优装载的  $x_i$  值。
2. 修改程序 15-1，使用一个表格来确定  $f(i,y)$  是否已被计算过。在求  $f(i,y)$  时，若表中已经存在该值，则直接取用；若不存在该值，则采用一个递归调用来计算该值。
3. 定义 0/1/2 背包问题为： $\max \sum_{i=1}^n p_i x_i$ 。限制条件为： $\sum_{i=1}^n w_i x_i \leq c$  且  $x_i \in \{0,1,2\}, 1 \leq i \leq n$ 。设  $f$  的定义同 0/1 背包问题中的定义。
  - 1) 从 0/1/2 背包问题中推出类似于 (15-1) 和 (15-2) 的公式。
  - 2) 假设  $ws$  为整数。编写一个类似于 15-2 的程序来计算二维数组  $f$ ，然后确定最优分配的  $x$  值。
  - 3) 程序的复杂性是多少？
4. 二维 0/1 背包问题定义为： $\max \sum_{i=1}^n p_i x_i$ 。限制条件为： $\sum_{i=1}^n v_i x_i \leq c, \sum_{i=1}^n w_i x_i \leq d$  且  $x_i \in \{0,1\}, 1 \leq i \leq n$ 。设  $f(i,y,z)$  为二维背包问题最优解的值，其中物品为  $i$  到  $n, c=y, d=z$ 。
  - 1) 推出类似于 (15-1) 和 (15-2) 式的关于  $f(n,y,z)$  和  $f(i,y,z)$  的公式。
  - 2) 假设  $vs$  和  $ws$  为整数。编写一个类似于 15-2 的程序计算三维数组  $f$ ，然后确定最优分配的  $x$  值。
  - 3) 程序的复杂性是多少？
- \*5. 编写一个实现元组方法的 C++ 代码，要求提供一个确定最优装载的  $x_i$  值的回溯函数。
6. 当取消段长限制时 (即在程序 15-3 中  $L = \infty$ )，程序 15-3 的时间复杂性按如下方式递归定义： $t(0) = c$  ( $c$  为常数)；当  $n > 0$  时  $t(n) = \sum_{i=0}^{n-1} t(i) + n$ 。
  - 1) 根据  $t(n-1) = \sum_{j=0}^{n-2} t(j) + n-1$  证明：当  $n > 0$  时， $t(n) = 2t(n-1) + 1$ 。
  - 2) 证明  $t(n) = \Theta(2^n)$



7. 编写函数Traceback (见程序15-3) 的迭代版本。试说明两个版本各自的优缺点。

8. 编写变长图像压缩过程中1) 和2) 的实现代码。

9. 证明： $\sum_{s=0}^{q-1} s(q-s+1) = \Theta(q^3)$ 。

10. 在求解矩阵乘法递归式时仅用到数组  $c$  和  $kay$  的上三角。重写程序 15-6, 定义  $c$  和  $kay$  为 UpperMatrix 类 (见 4.3.4 节) 的成员。

11. 改写程序 15-9, 把它作为 LinkedWDigraph 的类成员, 其渐进复杂性应与程序 15-9 相同。

12. 设  $G$  为有  $n$  个顶点的有向无环图,  $G$  中各顶点的编号为 1 到  $n$ , 且当  $i, j$  为  $G$  中的一条边时有  $i < j$ 。设  $l(i, j)$  为边  $i, j$  的长度:

1) 用动态规划方法计算图  $G$  中最长路径的长度, 算法的时间耗费应为  $O(h+e)$ , 其中  $e$  为  $G$  中的边数。

2) 编写一个函数, 利用 1) 中所得到的结果来构造最长路径, 其复杂性应为  $O(p)$ , 其中  $p$  为该路径的顶点数。

13. 改写程序 15-9, 首先从一个有向图的邻接矩阵开始, 然后计算其反身传递闭包矩阵 RTC。若从顶点  $i$  到顶点  $j$  无通路, 则  $RTC[i][j]=1$ , 否则  $RTC[i][j]=0$ 。要求代码的复杂性为  $\Theta(n^3)$ , 其中  $n$  为图中的顶点数。

14. 利用 (15-10) 式, 编写一个复杂性为  $O(n^2)$  的 C++ 迭代程序, 寻找等宽元件栈的最优折叠点。

15. 用递归式 (15-12) 代替式 (15-10) 完成练习 14, 时间复杂性要求为  $O(sn^2)$ 。

16. 用式 (15-13) 得出一个变宽元件栈的最小宽度折叠法, 时间复杂性要求为  $O(n^2)$ 。

17. 利用 15.2.6 节的设计, 给出一个寻找折叠矩形宽度为  $W$  的最小高度折叠算法, 其复杂性应为  $O(n^2 \log n)$ 。位 - 片元件宽度不等。

18. 利用式 (15-14) 和 (15-15) 来确定一个含  $n$  个标准单元的最小高度折叠。算法的时间复杂性应为  $O(n^2)$ 。能否使用这两个公式得到一个时间复杂性为  $\Theta(n)$  的算法?

\*19. 在 13.3.3 节可知, 一个工程可分解为多个任务且这些任务可按拓扑顺序来完成。设任务从 1 到  $n$  编号, 首先完成任务 1, 然后完成任务 2, 以此类推……。假设我们有两种方法来完成任务。  $C_{i,1}$  为使用第一种方法完成任务  $i$  时的代价,  $C_{i,2}$  为使用第 2 种方法完成任务  $i$  时的代价。令  $T_{i,1}$  为第一种方法中任务  $i$  的时间耗费,  $T_{i,2}$  为第二种方法中任务  $i$  的时间耗费。并设各个  $T$  为整数。设计一个动态规划算法, 以得到在时间  $t$  内完成所有任务的最小代价的方法。假定工程的代价为各任务的代价之和, 工程所需的时间是各任务时间耗费之和。(提示: 可设  $cost(i, j)$  为在  $j$  时间内完成任务  $i$  到  $n$  的最小代价)。算法的复杂性是多少?

\*20. 某一机器中有  $n$  个零件。每个零件有三个供应商, 来自供应商  $j$  的零件  $i$  的重量为  $W_{i,j}$ , 其价格为  $C_{i,j}$  ( $1 \leq j \leq 3$ )。机器的价格等于所有零件价格之和, 其重量也为各零件重量之和。设计一个动态规划算法, 以决定在总价格不超过  $C$  的条件下, 从哪些供应商购买零件能组成最轻的机器。(提示: 可设  $w(i, j)$  为价格低于  $j$  时由零件  $i$  到  $n$  组成的最轻机器)。算法的复杂性是多少?

\*21. 定义  $w(i, j)$  为价格低于  $j$  时由零件 1 到  $i$  组成的最轻机器, 完成练习 20。

\*22. 串  $s$  为串  $a$  中去掉某些字符而得到的子串。如串 “onion” 为串 “recognition” 的子串。当且仅当串  $s$  既是  $a$  的子串又是  $b$  的子串时, 串  $s$  是串  $a$  和串  $b$  的公共子串。串  $s$  的长度指其所含的字符数。试用动态规划算法得到串  $a$  和串  $b$  的最长公共子串。(提示: 设  $a=a_1a_2\dots a_n$ ,  $b=b_1b_2\dots b_m$ 。定义  $l(i, j)$  为串  $a_i\dots a_n$  和  $b_j\dots b_m$  最长公共子串的长度)。算法的复杂性是多少?

\*23. 若 $l(i,j)$ 定义为串 $a_1a_2\dots a_i$ 和 $b_1b_2\dots b_j$ 的最长公共子串的长度,重做练习22。

\*24. 在串编辑问题中,给出两个串 $a=a_1a_2\dots a_n$ 和 $b=b_1b_2\dots b_m$ 及三个耗费函数 $C$ ,  $D$ 和 $I$ 。其中 $C(i,j)$ 为将 $a_i$ 改为 $b_j$ 的耗费, $D(i)$ 为从 $a$ 中删除 $a_i$ 的耗费, $I(i)$ 为将 $b_i$ 插入 $a$ 中的耗费。通过修改、删除和插入操作可把串 $a$ 改为串 $b$ 。如,可删除所有 $a_i$ ,然后插入所有 $b_i$ ;或者当 $n \leq m$ 时,可先把 $a_i$ 变成 $b_i$  ( $1 \leq i \leq n$ ),然后删除其余的 $a_i$ 。整个操作序列的耗费为各个操作的耗费之和。设计一个动态规划算法来确定一个具有最少耗费的编辑操作序列。(提示:定义 $c(i,j)$ 为将 $a_1a_2\dots a_i$ 转变为 $b_1b_2\dots b_j$ 的最少耗费)。算法的复杂性是多少?

### 15.3 参考及推荐读物

1) V.Bhaskaran, K.Konstantinides. *Image and video Compression Standards*. Kluwer Academic, 1995。其中包含图像压缩算法的更多信息。

2) S.Sahni, B.Vemuri, F.Chen, C.kapoor, C.Leonard, J.Fitzsimmons. State of the Art LossLess Image Compression. Technical Report, University of Florida 1997。其中介绍了15.2.2节的变长模式。

3) T.Hu, M.Shing. Computation of Matrix Chain Products. 第1和第2部分。 *SLAM journal on Computing*, 11, 1982, 361~371 和13, 1984, 228~251。其中讲述了一个复杂性为 $O(n \log n)$ 的矩阵乘法链求解算法。

4) K.Supowit. Finding a Maximum Planar Subset of a Set of Nets in a Channel. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and systems*, 6, 1, 1987.93~94。其中无交叉子集的动态规划算法。

5) 以下论文介绍了位-片和标准单元折叠问题: D.Paik, S.Sahni. Optimal Folding of Bit Sliced Stacks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12, 11, 1993, 1679~1685; V.Thanvantri, S.Sahni. Folding a Stack of Equal Width Components. *IEEE Transactions on CAD of ICAS*, 14, 6, 1995, 775~780。其中提出的参数化搜索算法比本书中的动态规划算法更快; V.Thanvantri, S.Sahni. Optimal Folding of Standard and Custom Cells. *ACM Transactions on Design Automation and Electronic systems*, 1996。其中介绍了如何在 $O(n)$ 的时间内计算公式(15-14)和(15-15)。这三篇论文还考察了其他类型的折叠问题。



## 第16章 回溯

寻找问题的解的一种可靠的方法是首先列出所有候选解，然后依次检查每一个，在检查完所有或部分候选解后，即可找到所需要的解。理论上，当候选解数量有限并且通过检查所有或部分候选解能够得到所需解时，上述方法是可行的。不过，在实际应用中，很少使用这种方法，因为候选解的数量通常都非常大（比如指数级，甚至是大数阶乘），即便采用最快的计算机也只能解决规模很小的问题。

对候选解进行系统检查的方法有多种，其中回溯和分枝定界法是比较常用的两种方法。按照这两种方法对候选解进行系统检查通常会使问题的求解时间大大减少（无论对于最坏情形还是对于一般情形）。事实上，这些方法可以使我们避免对很大的候选解集合进行检查，同时能够保证算法运行结束时可以找到所需要的解。因此，这些方法通常能够用来求解规模很大的问题。

本章集中阐述回溯方法，这种方法被用来设计货箱装船、背包、最大完备子图、旅行商和电路板排列问题的求解算法。

### 16.1 算法思想

回溯（backtracking）是一种系统地搜索问题解答的方法。在5.5.6节求解迷宫老鼠问题时即采用了回溯技术。为了实现回溯，首先需要为问题定义一个解空间（solution space），这个空间必须至少包含问题的一个解（可能是最优的）。在迷宫老鼠问题中，我们可以定义一个包含从入口到出口的所有路径的解空间；在具有 $n$ 个对象的0/1背包问题中（见13.4节和15.2节），解空间的一个合理选择是 $2^n$ 个长度为 $n$ 的0/1向量的集合，这个集合表示了将0或1分配给 $x$ 的所有可能方法。当 $n=3$ 时，解空间为 $\{(0,0,0), (0,1,0), (0,0,1), (1,0,0), (0,1,1), (1,0,1), (1,1,0), (1,1,1)\}$ 。

下一步是组织解空间以便它能被容易地搜索。典型的组织方法是图或树。图16-1用图的形式给出了一个 $3 \times 3$ 迷宫的解空间。从 $(1,1)$ 点到 $(3,3)$ 点的每一条路径都定义了 $3 \times 3$ 迷宫解空间中的一个元素，但由于障碍的设置，有些路径是不可行的。

图16-2用树形结构给出了含三个对象的0/1背包问题的解空间。从 $i$ 层节点到 $i+1$ 层节点的一条边上的数字给出了向量 $x$ 中第 $i$ 个分量的值 $x_i$ ，从根节点到叶节点的每一条路径定义了解空间中的一个元素。从根节点A到叶节点H的路径定义了解 $x=[1,1,1]$ 。根据 $w$ 和 $c$ 的值，从根到叶的路径中的一些解或全部解可能是不可行的。

一旦定义了解空间的组织方法，这个空间即可按深度优先的方法从开始节点进

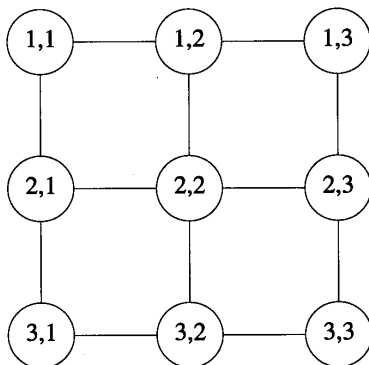


图16-1  $3 \times 3$ 迷宫的解空间

行搜索。在迷宫老鼠问题中，开始节点为入口节点 (1,1)；在0/1背包问题中，开始节点为根节点A。开始节点既是一个活节点又是一个E-节点 (expansion node)。从E-节点可移动到一个新节点。如果能从当前的E-节点移动到一个新节点，那么这个新节点将变成一个活节点和新的E-节点，旧的E-节点仍是一个活节点。如果不能移到一个新节点，当前的E-节点就“死”了 (即不再是一个活节点)，那么便只能返回到最近被考察的活节点 (回溯)，这个活节点变成了新的E-节点。当我们已经找到了答案或者回溯尽了所有的活节点时，搜索过程结束。

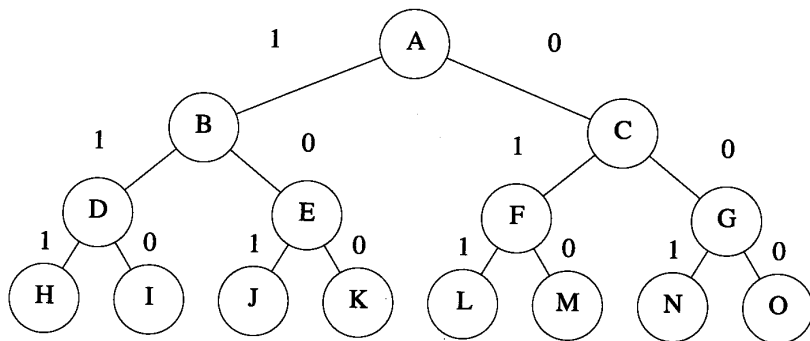


图16-2 三个对象的背包问题的解空间

例16-1 [迷宫老鼠] 考察图16-3a 的矩阵中给出的 $3 \times 3$ 的“迷宫老鼠”问题。我们将利用图16-1给出的解空间图来搜索迷宫。

从迷宫的入口到出口的每一条路径都与图16-1中从(1,1)到(3,3)的一条路径相对应。然而，图16-1中有些从(1,1)到(3,3)的路径却不是迷宫中从入口到出口的路径。

搜索从点(1,1)开始，该点是目前唯一的活节点，它也是一个E-节点。为避免再次走过这个位置，置 $maze(1,1)$ 为1。从这个位置，能移动到(1,2)或(2,1)两个位置。对于本例，两种移动都是可行的，因为在每一个位置都有一个值0。假定选择移动到(1,2)， $maze(1,2)$ 被置为1以避免再次经过该点。迷宫当前状态如图16-3b所示。这时有两个活节点(1,1) (1,2)。(1,2)成为E-节点。在图16-1中从当前E-节点开始有3个可能的移动，其中两个是不可行的，因为迷宫在这些位置上的值为1。唯一可行的移动是(1,3)。移动到这个位置，并置 $maze(1,3)$ 为1以避免再次经过该点，此时迷宫状态为16-3c。图16-1中，从(1,3)出发有两个可能的移动，但没有一个是可行的。所以E-节点(1,3)死亡，回溯到最近被检查的活节点(1,2)。在这个位置也没有可行的移动，故这个节点也死亡了。唯一留下的活节点是(1,1)。这个节点再次变为E-节点，它可移动到(2,1)。现在活节点为(1,1)，(2,1)。继续下去，能到达点(3,3)。此时，活节点表为(1,1)，(2,1)，(3,1)，(3,2)，(3,3)，这即是到达出口的路径。

程序5-13是一个在迷宫中寻找路径的回溯算法。

0 0 0

0 1 1

0 0 0

a)

1 1 0

0 1 1

0 0 0

b)

1 1 1

0 1 1

0 0 0

c)

图16-3 迷宫

例16-2 [0/1背包问题] 考察如下背包问题： $n=3$ ， $w=[20,15,15]$ ， $p=[40,25,25]$ 且 $c=30$ 。从根节点开始搜索图16-2中的树。根节点是当前唯一的活节点，也是E-节点，从这里能够移动到B或C点。假设移动到B，则活节点为A和B。B是当前E-节点。在节点B，剩下的容量 $r$ 为10，而收益 $cp$ 为40。从B点，能移动到D或E。移到D是不可行的，因为移到D所需的容量 $w_2$ 为15。到E的移动是可行的，因为在这个移动中没有占用任何容量。E变成新的E-节点。这时活节点为A,B,E。在节点E， $r=10$ ， $cp=40$ 。从E，有两种可能移动（到J和K），到J的移动是不可行的，而到K的移动是可行的。节点K变成了新的E-节点。因为K是一个叶子，所以得到一个可行的解。这个解的收益为 $cp=40$ 。 $x$ 的值由从根到K的路径来决定。这个路径（A,B,E,K）也是此时的活节点序列。既然不能进一步扩充K，K节点死亡，回溯到E，而E也不能进一步扩充，它也死亡了。

接着，回溯到B，它也死亡了，A再次变为E-节点。它可被进一步扩充，到达节点C。此时 $r=30$ ， $cp=0$ 。从C点能够移动到F或G。假定移动到F。F变为新的E-节点并且活节点为A，C，F。在F， $r=15$ ， $cp=25$ 。从F点，能移动到L或M。假定移动到L。此时 $r=0$ ， $cp=50$ 。既然L是一个叶节点，它表示了一个比目前找到的最优解（即节点K）更好的可行解，我们把这个解作为最优解。节点L死亡，回溯到节点F。继续下去，搜索整棵树。在搜索期间发现的最优解即为最后的解。

例16-3 [旅行商问题] 在这个问题中，给出一个 $n$ 顶点网络（有向或无向），要求找出一个包含所有 $n$ 个顶点的具有最小耗费的环路。任何一个包含网络中所有 $n$ 个顶点的环路被称作一个旅行（tour）。在旅行商问题中，要设法找到一条最小耗费的旅行。

图16-4给出了一个四顶点网络。在这个网络中，一些旅行如下： $1,2,4,3,1$ ； $1,3,2,4,1$ 和 $1,4,3,2,1$ 。旅行 $2,4,3,1,2$ ； $4,3,1,2,4$ 和 $3,1,2,4,3$ 和旅行 $1,2,4,3,1$ 一样。而旅行 $1,3,4,2,1$ 是旅行 $1,2,4,3,1$ 的“逆”。旅行 $1,2,4,3,1$ 的耗费为66；而 $1,3,2,4,1$ 的耗费为25； $1,4,3,2,1$ 为59。故 $1,3,2,4,1$ 是该网络中最小耗费的旅行。

顾名思义，旅行商问题可被用来模拟现实生活中旅行商所要旅行的地区问题。顶点表示旅行商所要旅行的城市（包括起点）。边的耗费给出了在两个城市旅行所需的时间（或花费）。旅行表示当旅行商游览了所有城市再回到出发点时所走的路线。

旅行商问题还可用来模拟其他问题。假定要在一个金属薄片或印刷电路板上钻许多孔。孔的位置已知。这些孔由一个机器钻头来钻，它从起始位置开始，移动到每一个钻孔位置钻孔，然后回到起始位置。总共花的时间是钻所有孔的时间与钻头移动的时间。钻所有孔所需的时间独立于钻孔顺序。然而，钻头移动时间是钻头移动距离的函数。因此，希望找到最短的移动路径。

另有一个例子，考察一个批量生产的环境，其中有一个特殊的机器可用来生产 $n$ 个不同的产品。利用一个生产循环不断地生产这些产品。在一个循环中，所有 $n$ 个产品被顺序生产出来，然后再开始下一个循环。在下一个循环中，又采用了同样的生产顺序。例如，如果这台机器被用来顺序为小汽车喷红、白、蓝漆，那么在为蓝色小汽车喷漆之后，我们又开始了新一轮循环，为红色小汽车喷漆，然后是白色小汽车、蓝色小汽车、红色小汽车，……，如此下去。一个循

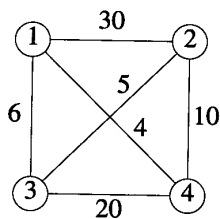


图16-4 一个四顶点网络



环的花费包括生产一个循环中的产品所需的花费以及循环中从一个产品转变到另一个产品的花费。虽然生产产品的花费独立于产品生产顺序，但循环中从生产一个产品转变到生产另一个产品的花费却与顺序有关。为了使耗费最小化，可以定义一个有向图，图中的顶点表示产品，边  $\langle i, j \rangle$  上的耗费值为生产过程中从产品  $i$  转变到产品  $j$  所需的耗费。一个最小耗费的旅行定义了一个最小耗费的生产品循环。

既然旅行是包含所有顶点的一个循环，故可以把任意一个点作为起点（因此也是终点）。针对图16-4，任意选取点1作为起点和终点，则每一个旅行可用顶点序列  $1, v_2, \dots, v_n, 1$  来描述， $v_2, \dots, v_n$  是  $(2, 3, \dots, n)$  的一个排列。可能的旅行可用一个树来描述，其中每一个从根到叶的路径定义了一个旅行。图16-5给出了一棵表示四顶点网络的树。从根到叶的路径中各边的标号定义了一个旅行（还要附加1作为终点）。例如，到节点L的路径表示了旅行1,2,3,4,1，而到节点O的路径表示了旅行1,3,4,2,1。网络中的每一个旅行都由树中的一条从根到叶的确定路径来表示。因此，树中叶的数目为  $(n-1)!$ 。

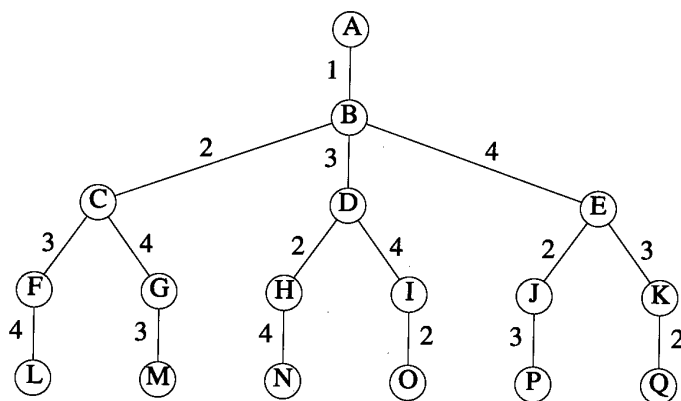


图16-5 四顶点网络的解空间树

回溯算法将用深度优先方式从根节点开始，通过搜索解空间树发现一个最小耗费的旅行。对图16-4的网络，利用图16-5的解空间树，一个可能的搜索为 ABCFL。在L点，旅行1,2,3,4,1作为当前最好的旅行被记录下来。它的耗费是59。从L点回溯到活节点F。由于F没有未被检查的孩子，所以它成为死节点，回溯到C点。C变为E-节点，向前移动到G，然后是M。这样构造出了旅行1,2,4,3,1，它的耗费是66。既然它不比当前的最佳旅行好，抛弃它并回溯到G，然后是C,B。从B点，搜索向前移动到D，然后是H,N。这个旅行1,3,2,4,1的耗费是25，比当前的最佳旅行好，把它作为当前的最好旅行。从N点，搜索回溯到H，然后是D。在D点，再次向前移动，到达O点。如此继续下去，可搜索完整棵树，得出1,3,2,4,1是最少耗费的旅行。

当要求解的问题需要根据  $n$  个元素的一个子集来优化某些函数时，解空间树被称作子集树 (subset tree)。所以对有  $n$  个对象的0/1背包问题来说，它的解空间树就是一个子集树。这样一棵树有  $2^n$  个叶节点，全部节点有  $2^{n+1} - 1$  个。因此，每一个对树中所有节点进行遍历的算法都必须耗时  $(2^n)$ 。当要求解的问题需要根据一个  $n$  元素的排列来优化某些函数时，解空间树被称作排列树 (permutation tree)。这样的树有  $n!$  个叶节点，所以每一个遍历树中所有节点的算法都必须耗时  $(n!)$ 。图16-5中的树是顶点  $\{2,3,4\}$  的最佳排列的解空间树，顶点1是旅行的起点和终点。



通过确定一个新近到达的节点能否导致一个比当前最优解还要好的解，可加速对最优解的搜索。如果不能，则移动到该节点的任何一个子树都是无意义的，这个节点可被立即杀死，用来杀死活节点的策略称为限界函数（bounding function）。在例16-2中，可使用如下限界函数：杀死代表不可行解决方案的节点；对于旅行商问题，可使用如下限界函数：如果目前建立的部分旅行的耗费不少于当前最佳路径的耗费，则杀死当前节点。如果在图16-4的例子中使用该限界函数，那么当到达节点I时，已经找到了具有耗费25的1,3,2,4,1的旅行。在节点I，部分旅行1,3,4的耗费为26，若旅行通过该节点，那么不能找到一个耗费小于25的旅行，故搜索以I为根节点的子树毫无意义。

小结

回溯方法的步骤如下：

- 1) 定义一个解空间，它包含问题的解。
- 2) 用适于搜索的方式组织该空间。
- 3) 用深度优先法搜索该空间，利用限界函数避免移动到不可能产生解的子空间。

回溯算法的一个有趣的特性是在搜索执行的同时产生解空间。在搜索期间的任何时刻，仅保留从开始节点到当前E-节点的路径。因此，回溯算法的空间需求为 $O$ （从开始节点起最长路径的长度）。这个特性非常重要，因为解空间的大小通常是最长路径长度的指数或阶乘。所以如果要存储全部解空间的话，再多的空间也不够用。

## 练习

1. 考察如下0/1背包问题： $n=4$ ， $w=[20,25,15,35]$ ， $p=[40,49,25,60]$ ， $c=62$ 。

1) 画出该0/1背包问题的解空间树。

2) 对该树运用回溯算法（利用给出的 $ps, ws, c'$ 值），依回溯算法遍历节点的顺序标记节点。确定回溯算法未遍历的节点。

2. 1) 当 $n=5$ 时，画出旅行商问题的解空间树。

2) 在该树上，运用回溯算法（使用图16-6的例子）。依回溯算法遍历节点的顺序标记节点。确定未被遍历的节点。

3. 每周六，Mary 和Joe 都在一起打乒乓球。她们每人都有一个装有120个球的篮子。这样一直打下去，直到两个篮子为空。然后她们需要从球桌周围拾起240个球，放入各自的篮子。Mary 只拾她这边的球，而Joe 拾剩下的球。描述如何用旅行商问题帮助 Mary 和Joe 决定她们拾球的顺序以便她们能走最少的路径。

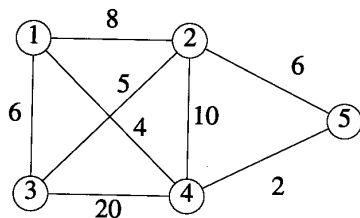


图16-6 练习2的实例

## 16.2 应用

### 16.2.1 货箱装船

#### 1. 问题

在13.3节中，考察了用最大数量的货箱装船的问题。现在对该问题做一些改动。在新问题中，有两艘船， $n$ 个货箱。第一艘船的载重量是 $c_1$ ，第二艘船的载重量是 $c_2$ ， $w_i$ 是货箱 $i$ 的重量

且  $\sum_{i=1}^n w_i = c_1 + c_2$ 。我们希望确定是否有一种可将所有  $n$  个货箱全部装船的方法。若有的话,找出该方法。

例16-4 当  $n=3$ ,  $c_1=c_2=50$ ,  $w=[10,40,40]$  时,可将货箱1,2装到第一艘船上,货箱3装到第二艘船上。如果  $w=[20,40,40]$ ,则无法将货箱全部装船。

当  $\sum_{i=1}^n w_i = c_1 + c_2$  时,两艘船的装载问题等价于子集之和 (sum-of-subset) 问题,即有  $n$  个数字,要求找到一个子集 (如果存在的话) 使它的和为  $c_1$ 。当  $c_1=c_2$  且  $\sum_{i=1}^n w_i = 2c_1$  时,两艘船的装载问题等价于分割问题 (partition problem),即有  $n$  个数字  $a_i$ ,  $(1 \leq i \leq n)$ ,要求找到一个子集 (若存在的话),使得子集之和为  $(\sum_{i=1}^n a_i)/2$ 。分割问题和子集之和的问题都是 NP-复杂问题。而且即使问题被限制为整型数字,它们仍是 NP-复杂问题。所以不能期望在多项式时间内解决两艘船的装载问题。

当存在一种方法能够装载所有  $n$  个货箱时,可以验证以下的装船策略可以获得成功: 1) 尽可能地将第一艘船装至它的重量极限; 2) 将剩余货箱装到第二艘船。为了尽可能地将第一艘船装满,需要选择一个货箱的子集,它们的总重量尽可能接近  $c_1$ 。这个选择可通过求解 0/1 背包问题来实现,即寻找  $\max (\sum_{i=1}^n w_i x_i)$ , 其中  $\sum_{i=1}^n w_i x_i \leq c_1$ ,  $x_i \in \{0, 1\}$ ,  $1 \leq i \leq n$ 。

当重量是整数时,可用 15.2 节的动态规划方法确定第一艘船的最佳装载。用元组方法所需时间为  $O(\min\{c_1, 2^n\})$ 。可以使用回溯方法设计一个复杂性为  $O(2^n)$  的算法,在有些实例中,该方法比动态规划算法要好。

## 2. 第一种回溯算法

既然想要找到一个重量的子集,使子集之和尽量接近  $c_1$ ,那么可以使用一个子集空间,并将其组织成如图 16-2 那样的二叉树。可用深度优先的方法搜索该解空间以求得最优解。使用限界函数去阻止不可能获得解答的节点的扩张。如果  $Z$  是树的  $j+1$  层的一个节点,那么从根到  $O$  的路径定义了  $x_i$  ( $1 \leq i \leq j$ ) 的值。使用这些值,定义  $cw$  (当前重量) 为  $\sum_{i=1}^n w_i x_i$ 。若  $cw > c_1$ ,则以  $O$  为根的子树不能产生一个可行的解答。可将这个测试作为限界函数。当且仅当一个节点的  $cw$  值大于  $c_1$  时,定义它是不可行的。

例16-5 假定  $n=4$ ,  $w=[8,6,2,3]$ ,  $c_1=12$ 。解空间树为图 16-2 的树再加上一层节点。搜索从根  $A$  开始且  $cw=0$ 。若移动到左孩子  $B$  则  $cw=8$ ,  $cw \leq c_1=12$ 。以  $B$  为根的子树包含一个可行的节点,故移动到节点  $B$ 。从节点  $B$  不能移动到节点  $D$ , 因为  $cw+w_2 > c_1$ 。移动到节点  $E$ , 这个移动未改变  $cw$ 。下一步为节点  $J$ ,  $cw=10$ 。  $J$  的左孩子的  $cw$  值为 13, 超出了  $c_1$ , 故搜索不能移动到  $J$  的左孩子。可移动到  $J$  的右孩子,它是一个叶节点。至此,已找到了一个子集,它的  $cw=10$ 。  $x_i$  的值由从  $A$  到  $J$  的右孩子的路径获得,其值为  $[1,0,1,0]$ 。

回溯算法接着回溯到  $J$ , 然后是  $E$ 。从  $E$ , 再次沿着树向下移动到节点  $K$ , 此时  $cw=8$ 。移动到它的左子树,有  $cw=11$ 。既然已到达了一个叶节点,就看是否  $cw$  的值大于当前的最优  $cw$  值。结果确实大于最优值,所以这个叶节点表示了一个比  $[1,0,1,0]$  更好的解决方案。到该节点的路径决定了  $x$  的值  $[1,0,0,1]$ 。

从该叶节点,回溯到节点  $K$ , 现在移动到  $K$  的右孩子,一个具有  $cw=8$  的叶节点。这个叶节点中没有比当前最优  $cw$  值还好的  $cw$  值,所以回溯到  $K, E, B$  直到  $A$ 。从根节点开始,沿树继续向

下移动。算法将移动到C并搜索它的子树。

当使用前述的限界函数时，便产生了程序 16-1所示的回溯算法。函数 MaxLoading返回 c 的最大子集之和，但它不能找到产生该和的子集。后面将改进代码以便找到这个子集。MaxLoading调用了递归函数maxLoading，它是类Loading的一个成员，定义Loading类是为了减少MaxLoading中的参数个数。maxLoading(1) 实际执行解空间的搜索。MaxLoading(i) 搜索以i层节点（该节点已被隐式确定）为根的子树。从根到该节点的路径定义的子解答有一个重量值cw，目前最优解答的重量为bestw，这些变量以及与类Loading的一个成员相关联的其他变量，均由MaxLoading初始化。

程序 16-1 第一种回溯算法

```
template<class T>
class Loading {
    friend MaxLoading(T [], T, int);
private:
    void maxLoading(int i);
    int n;      // 货箱数目
    T *w,       // 货箱重量数组
    c,          // 第一艘船的容量
    cw,         // 当前装载的重量
    bestw;      // 目前最优装载的重量
};
```

```
template<class T>
void Loading<T>::maxLoading(int i)
{// 从第 i 层节点搜索
    if (i > n) { // 位于叶节点
        if (cw > bestw) bestw = cw;
        return;}
    // 检查子树
    if (cw + w[i] <= c) { // 尝试x[i] = 1
        cw += w[i];
        maxLoading(i+1);
        cw -= w[i];}
    maxLoading(i+1); // 尝试x[i] = 0
}
```

```
template<class T>
T MaxLoading(T w[], T c, int n)
{// 返回最优装载的重量
    Loading<T> X;
    // 初始化X
    X.w = w;
    X.c = c;
    X.n = n;
    X.bestw = 0;
    X.cw = 0;

    // 计算最优装载的重量
```

```

X.maxLoading(1);
return X.bestw;
}

```

如果  $i > n$ ，则到达了叶节点。被叶节点定义的解答有重量  $cw$ ，它一定  $\leq c$ ，因为搜索不会移动到不可行的节点。若  $cw > bestw$ ，则目前最优解答的值被更新。当  $i = n$  时，我们处在有两个孩子的节点  $Z$  上。左孩子表示  $x[i]=1$  的情况，只有  $cw + w[i] \leq c$  时，才能移到这里。当移动到左孩子时， $cw$  被置为  $cw + w[i]$ ，且到达一个  $i+1$  层的节点。以该节点为根的子树被递归搜索。当搜索完成时，回到节点  $Z$ 。为了得到  $Z$  的  $cw$  值，需用当前的  $cw$  值减去  $w[i]$ ， $Z$  的右子树还未搜索。既然这个子树表示  $x[i]=0$  的情况，所以无需进行可行性检查就可移动到该子树，因为一个可行节点的右孩子总是可行的。

注意：解空间树未被 `maxLoading` 显示构造。函数 `maxLoading` 在它到达的每一个节点上花费  $\Theta(1)$  时间。到达的节点数量为  $O(2^n)$ ，所以复杂性为  $O(2^n)$ 。这个函数使用的递归栈空间为  $\Theta(n)$ 。

### 3. 第二种回溯方法

通过不移动到不可能包含比当前最优解还要好的解的右子树，能提高函数 `maxLoading` 的性能。令 `bestw` 为目前最优解的重量， $Z$  为解空间树的第  $i$  层的一个节点， $cw$  的定义如前。以  $Z$  为根的子树中没有叶节点的重量会超过  $cw + r$ ，其中  $r = \sum_{j=i+1}^n w[j]$  为剩余货箱的重量。因此，当  $cw + r \leq bestw$  时，没有必要去搜索  $Z$  的右子树。

例16-6 令  $n, w, c_i$  的值与例16-5中相同。用新的限界函数，搜索将像原来那样向前进行直至到达第一个叶节点  $J$ （它是  $J$  的右孩子）。`bestw` 被置为10。回溯到  $E$ ，然后向下移动到  $K$  的左孩子，此时 `bestw` 被更新为11。我们没有移动到  $K$  的右孩子，因为在右孩子节点  $cw=8, r=0, cw+r \leq bestw$ 。回溯到节点  $A$ 。同样，不必移动到右孩子  $C$ ，因为在  $C$  点  $cw=0, r=11$  且  $cw+r > bestw$ 。

加强了条件的限界函数避免了对  $A$  的右子树及  $K$  的右子树的搜索。

当使用加强了条件的限界函数时，可得到程序 16-2 的代码。这个代码将类型为  $T$  的私有变量  $r$  加到了类 `Loading` 的定义中。新的代码不必检查是否一个到达的叶节点有比当前最优解还优的重量值。这样的检查是不必要的，因为加强的限界函数不允许移动到不能产生较好解的节点。因此，每到达一个新的叶节点就意味着找到了比当前最优解还优的解。虽然新代码的复杂性仍是  $O(2^n)$ ，但它可比程序 16-1 少搜索一些节点。

程序 16-2 程序 16-1 的优化

```

template<class T>
void Loading<T>::maxLoading(int i)
{// // 从第 i 层节点搜索
    if (i > n) { // 在叶节点上
        bestw = cw;
        return;}
    // 检查子树
    r -= w[i];
    if (cw + w[i] <= c) { // 尝试  $x[i] = 1$ 
        cw += w[i];
        maxLoading(i+1);
    }
}

```

```

    cw -= w[i];}
    if (cw + r > bestw) //尝试x[i] = 0
        maxLoading(i+1);
    r += w[i];
}

```

```

template<class T>
T MaxLoading(T w[], T c, int n)
{// 返回最优装载的重量
    Loading<T> X;
    // 初始化 X
    X.w = w;
    X.c = c;
    X.n = n;
    X.bestw = 0;
    X.cw = 0;
    // r的初始值为所有重量之和
    X.r = 0;
    for (int i = 1; i <= n; i++)
        X.r += w[i];

    // 计算最优装载的重量
    X.maxLoading(1);
    return X.bestw;
}

```

#### 4. 寻找最优子集

为了确定具有最接近  $c$  的重量的货箱子集，有必要增加一些代码来记录当前找到的最优子集。为了记录这个子集，将参数  $bestx$  添加到  $Maxloading$  中。 $bestx$  是一个整数数组，其中元素可为0或1，当且仅当  $bestx[i]=1$  时，货箱  $i$  在最优子集中。新的代码见程序 16-3。

程序 16-3 给出最优装载的代码

```

template<class T>
void Loading<T>::maxLoading(int i)
{//从第 i 层节点搜索
    if (i > n) {// 在叶节点上
        for (int j = 1; j <= n; j++)
            bestx[j] = x[j];
        bestw = cw; return;}
    // 检查子树
    r -= w[i];
    if (cw + w[i] <= c) {//尝试 x[i] = 1
        x[i] = 1;
        cw += w[i];
        maxLoading(i+1);
        cw -= w[i];}
    if (cw + r > bestw) {//尝试x[i] = 0
        x[i] = 0;
        maxLoading(i+1);}
}

```

```

    r += w[i];
}

template<class T>
T MaxLoading(T w[], T c, int n, int bestx[])
{// 返回最优装载及其值
    Loading<T> X;
    // 初始化 X
    X.x = new int [n+1];
    X.w = w;
    X.c = c;
    X.n = n;
    X.bestx = bestx;
    X.bestw = 0;
    X.cw = 0;
    // r的初始值为所有重量之和
    X.r = 0;
    for (int i = 1; i <= n; i++)
        X.r += w[i];
    X.maxLoading(1);
    delete [] X.x;
    return X.bestw;
}

```

这段代码在Loading中增加了两个私有数据成员： $x$  和  $bestx$ 。这两个私有数据成员都是整型的一维数组。数组  $x$  用来记录从搜索树的根到当前节点的路径（即它保留了路径上的  $x_i$  值）， $bestx$  记录当前最优解。无论何时到达了一个具有较优解的叶节点， $bestx$  被更新以表示从根到叶的路径。为1的  $x_i$  值确定了要被装载的货箱。数据  $x$  的空间由MaxLoading 分配。

因为  $bestx$  可以被更新  $O(2^n)$  次，故  $maxLoading$  的复杂性为  $O(n2^n)$ 。使用下列方法之一，复杂性可降为  $O(2^n)$ ：

1) 首先运行程序 16-2 的代码，以决定最优装载重量，令其为  $W$ 。然后运行程序 16-3 的一个修改版本。该版本以  $bestw=W$  开始运行，当  $cw+r \geq bestw$  时搜索右子树，第一次到达一个叶节点时便终止（即  $i>n$ ）。

2) 修改程序 16-3 的代码以不断保留从根到当前最优叶的路径。尤其当位于  $i$  层节点时，则到最优叶的路径由  $x[j]$  ( $1 \leq j < i$ ) 和  $bestx[j]$  ( $j \leq i \leq n$ ) 给出。按照这种方法，算法每次回溯一级，并在  $bestx$  中存储一个  $x[i]$ 。由于算法回溯所需时间为  $O(2^n)$ ，因此额外开销为  $O(2^n)$ 。

##### 5. 一个改进的迭代版本

可改进程序 16-3 的代码以减少它的空间需求。因为数组  $x$  中记录可在树中移动的所有路径，故可以消除大小为  $\Theta(n)$  的递归栈空间。如例 16-5 所示，从解空间树的任何节点，算法不断向左孩子移动，直到不能再移动为止。如果一个叶子已被到达，则最优解被更新。否则，它试图移动到右孩子。当要么到达一个叶节点，要么不值得移动到一个右孩子时，算法回溯到一个节点，条件是从该节点向其右孩子移动有可能找到最优解。这个节点有一个特性，即它是路径中具有  $x[i]=1$  的节点中离根节点最近的节点。如果向右孩子的移动是有效的，那么就这么做，然后再完成一系列向左孩子的移动。如果向右孩子的移动是无效的，则回溯到  $x[i]=1$  的下一个节点。该算法遍历树的方式可被编码成与程序 16-4 一样的迭代（即循环）算法。不像递归代码，这种

代码在检查是否该向右孩子移动之前就移动到了右孩子。如果这个移动是不可行的，则回溯。迭代代码的时间复杂性与程序 16-3 一样。

程序 16-4 迭代代码

```
template<class T>
T MaxLoading(T w[], T c, int n, int bestx[])
{// 返回最佳装载及其值
// 迭代回溯程序
// 初始化根节点
int i = 1; // 当前节点的层次
// x[1:i-1] 是到达当前节点的路径
int *x = new int [n+1];
T bestw = 0, // 迄今最优装载的重量
    cw = 0, // 当前装载的重量
    r = 0; // 剩余货箱重量的和
for (int j = 1; j <= n; j++)
    r += w[j];

// 在树中搜索
while (true) {
    // 下移，尽可能向左
    while (i <= n && cw + w[i] <= c) {
        // 移向左孩子
        r -= w[i];
        cw += w[i];
        x[i] = 1;
        i++;
    }

    if (i > n) { // 到达叶子
        for (int j = 1; j <= n; j++)
            bestx[j] = x[j];
        bestw = cw;
    }
    else { // 移向右孩子
        r -= w[i];
        x[i] = 0;
        i++;
    }

    // 必要时返回
    while (cw + r <= bestw) {
        // 本子树没有更好的叶子，返回
        i--;
        while (i > 0 && !x[i]) {
            // 从一个右孩子返回
            r += w[i];
            i--;
        }
    }

    if (i == 0) { delete [] x;
```



```

        return bestw;}

// 移向右子树
x[i] = 0;
cw -= w[i];
i++;
}
}
}

```

### 16.2.2 0/1 背包问题

0/1背包问题是一个NP-复杂问题，为了解决该问题，在13.4节采用了贪婪算法，在15.2节又采用了动态规划算法。在本节，将用回溯算法解决该问题。既然想选择一个对象的子集，将它们装入背包，以便获得的收益最大，则解空间应组织成子集树的形状（如图16-2所示）。该回溯算法与16.2节的装载问题很类似。首先形成一个递归算法，去找到可获得的最大收益。然后，对该算法加以改进，形成代码。改进后的代码可找到获得最大收益时包含在背包中的对象的集合。

与程序16-2一样，左孩子表示一个可行的节点，无论何时都要移动到它；当右子树可能含有比当前最优解还优的解时，移动到它。一种决定是否要移动到右子树的简单方法是令 $r$ 为还未遍历的对象的收益之和，将 $r$ 加到 $cp$ （当前节点所获收益）之上，若 $(r+cp) > bestp$ （目前最优解的收益），则不需搜索右子树。一种更有效的方法是按收益密度 $p_i/w_i$ 对剩余对象排序，将对象按密度递减的顺序去填充背包的剩余容量，当遇到第一个不能全部放入背包的对象时，就使用它的一部分。

**例16-7** 考察一个背包例子： $n=4$ ， $c=7$ ， $p=[9,10,7,4]$ ， $w=[3,5,2,1]$ 。这些对象的收益密度为 $[3,2,3.5,4]$ 。当背包以密度递减的顺序被填充时，对象4首先被填充，然后是对象3、对象1。在这三个对象被装入背包之后，剩余容量为1。这个容量可容纳对象2的0.2倍的重量。将0.2倍的该对象装入，产生了收益值2。被构造的解为 $x=[1,0.2,1,1]$ ，相应的收益值为22。尽管该解不可行（ $x_2$ 是0.2，而实际上它应为0或1），但它的收益值22一定不少于要求的最优解。因此，该0/1背包问题没有收益值多于22的解。

解空间树为图16-2再加上一层节点。当位于解空间树的节点B时， $x_1=1$ ，目前获益为 $cp=9$ 。该节点所用容量为 $cw=3$ 。要获得最好的附加收益，要以密度递减的顺序填充剩余容量 $cleft=c-cw=4$ 。也就是说，先放对象4，然后是对象3，然后是对象2的0.2倍的重量。因此，子树A的最优解的收益值至多为22。

当位于节点C时， $cp=cw=0$ ， $cleft=7$ 。按密度递减顺序填充剩余容量，则对象4和3被装入。然后是对象2的0.8倍被装入。这样产生出收益值19。在子树C中没有节点可产生出比19还大的收益值。

在节点E， $cp=9$ ， $cw=3$ ， $cleft=4$ 。仅剩对象3和4要被考虑。当对象按密度递减的顺序被考虑时，对象4先被装入，然后是对象3。所以在子树E中无节点有多于 $cp+4+7=20$ 的收益值。如果已经找到了一个具有收益值20或更多的解，则无必要去搜索E子树。

一种实现限界函数的好方法是首先将对象按密度排序。假定已经做了这样的排序。定义类Knap（见程序16-5）来减少限界函数Bound（见程序16-6）及递归函数Knapsack（见程序16-7）

的参数数量，该递归函数用于计算最优解的收益值。参数的减少又可引起递归栈空间的减少以及每一个Knapsack的执行时间的减少。注意函数Knapsack和函数maxLoading（见程序16-2）的相似性。同时注意仅当向右孩子移动时，限界函数才被计算。当向左孩子移动时，左孩子的限界函数的值与其父节点相同。

程序16-5 Knap类

```
template<class Tw, class Tp>
class Knap {
    friend Tp Knapsack(Tp *, Tw *, Tw, int);
private:
    Tp Bound(int i);
    void Knapsack(int i);
    Tw c;      // 背包容量
    int n;     // 对象数目
    Tw *w;     // 对象重量的数组
    Tp *p;     // 对象收益的数组
    Tw cw;     // 当前背包的重量
    Tp cp;     // 当前背包的收益
    Tp bestp;  // 迄今最大的收益
};
```

程序16-6 限界函数

```
template<class Tw, class Tp>
Tp Knap<Tw, Tp>::Bound(int i)
// 返回子树中最优叶子的上限值 Return upper bound on value of
// best leaf in subtree.
{
    Tw cleft = c - cw; // 剩余容量
    Tp b = cp;         // 收益的界限
    // 按照收益密度的次序装填剩余容量
    while (i <= n && w[i] <= cleft) {
        cleft -= w[i];
        b += p[i];
        i++;
    }

    // 取下一个对象的一部分
    if (i <= n) b += p[i]/w[i] * cleft;
    return b;
}
```

程序16-7 0/1背包问题的迭代函数

```
template<class Tw, class Tp>
void Knap<Tw, Tp>::Knapsack(int i)
// 从第 i 层节点搜索
{
    if (i > n) { // 在叶节点上
        bestp = cp;
    }
}
```

```

    return;}
// 检查子树
if (cw + w[i] <= c) {尝试x[i] = 1
    cw += w[i];
    cp += p[i];
    Knapsack(i+1);
    cw -= w[i];
    cp -= p[i];}
if (Bound(i+1) > bestp) // 尝试x[i] = 0
    Knapsack(i+1);
}

```

在执行程序 16-7 的函数 Knapsack 之前，需要按密度对对象排序，也要确保对象的重量总和超出背包的容量。为了完成排序，定义了类 Object（见程序 16-8）。注意定义操作符  $\leq$  是为了使归并排序程序（见程序 14-3）能按密度递减的顺序排序。

程序 16-8 Object 类

```

class Object {
    friend int Knapsack(int *, int *, int, int);
public:
    int operator<=(Object a) const
    {return (d >= a.d);}
private:
    int ID; // 对象号
    float d; // 收益密度
};

```

程序 16-9 首先验证重量之和超出背包容量，然后排序对象，在执行 Knap::Knapsack 之前完成一些必要的初始化。Knap::Knapsack 的复杂性是  $O(n^2)$ ，因为限界函数的复杂性为  $O(n)$ ，且该函数在  $O(2^n)$  个右孩子处被计算。

程序 16-9 程序 16-7 的预处理代码

```

template<class Tw, class Tp>
Tp Knapsack(Tp p[], Tw w[], Tw c, int n)
{// 返回最优装包的值
    // 初始化
    Tw W = 0; // 记录重量之和
    Tp P = 0; // 记录收益之和
    // 定义一个按收益密度排序的对象数组
    Object *Q = new Object [n];

    for (int i = 1; i <= n; i++) {
        // 收益密度的数组
        Q[i-1].ID = i;
        Q[i-1].d = 1.0*p[i]/w[i];
        P += p[i];
        W += w[i];
    }
}

```

```
if (W <= c) return P; // 可容纳所有对象
```

```
MergeSort(Q,n); // 按密度排序
```

```
// 创建Knap的成员
```

```
Knap<Tw, Tp> K;
```

```
K.p = new Tp [n+1];
```

```
K.w = new Tw [n+1];
```

```
for (i = 1; i <= n; i++) {
```

```
    K.p[i] = p[Q[i-1].ID];
```

```
    K.w[i] = w[Q[i-1].ID];
```

```
}
```

```
K.cp = 0;
```

```
K.cw = 0;
```

```
K.c = c;
```

```
K.n = n;
```

```
K.bestp = 0;
```

```
// 寻找最优收益
```

```
K.Knapsack(1);
```

```
delete [] Q;
```

```
delete [] K.w;
```

```
delete [] K.p;
```

```
return K.bestp;
```

```
}
```

### 16.2.3 最大完备子图

令 $U$ 为无向图 $G$ 的顶点的子集，当且仅当对于 $U$ 中的任意点 $u$ 和 $v$ ， $(u,v)$ 是图 $G$ 的一条边时， $U$ 定义了一个完全子图（complete subgraph）。子图的尺寸为图中顶点的数量。当且仅当一个完全子图不被包含在 $G$ 的一个更大的完全子图中时，它是图 $G$ 的一个完备子图。最大的完备子图是具有最大尺寸的完备子图。

例16-8 在图16-7a中，子集 $\{1,2\}$ 定义了一个尺寸为2的完全子图。这个子图不是一个完备子图，因为它被包含在一个更大的完全子图 $\{1,2,5\}$ 中。 $\{1,2,5\}$ 定义了该图的一个最大的完备子图。点集 $\{1,4,5\}$ 和 $\{2,3,5\}$ 定义了其他的最大的完备子图。

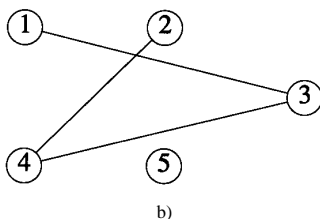
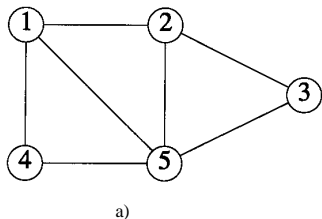


图16-7 图及其补图

a) 图 $G$  b) 补图 $\bar{G}$

当且仅当对于  $U$  中任意点  $u$  和  $v$ ,  $(u,v)$  不是  $G$  的一条边时,  $U$  定义了一个空子图。当且仅当一个子集不被包含在一个更大的点集中时, 该点集是图  $G$  的一个独立集 (independent set), 同时它也定义了图  $G$  的空子图。最大独立集是具有最大尺寸的独立集。对于任意图  $G$ , 它的补图 (complement)  $\bar{G}$  是有同样点集的图, 且当且仅当  $(u,v)$  不是  $G$  的一条边时, 它是  $\bar{G}$  的一条边。

例16-9 图16-7b 是图16-7a 的补图, 反之亦然。 $\{2,4\}$  定义了16-7a 的一个空子图, 它也是该图的一个最大独立集。虽然  $\{1,2\}$  定义了图16-7b 的一个空子图, 它不是一个独立集, 因为它被包含在空子图  $\{1,2,5\}$  中。 $\{1,2,5\}$  是图16-7b 中的一个最大独立集。

如果  $U$  定义了  $G$  的一个完全子图, 则它也定义了  $\bar{G}$  的一个空子图, 反之亦然。所以在  $G$  的完备子图与  $\bar{G}$  的独立集之间有对应关系。特别的,  $G$  的一个最大完备子图定义了  $\bar{G}$  的一个最大独立集。

最大完备子图问题是指寻找图  $G$  的一个最大完备子图。类似地, 最大独立集问题是指寻找图  $G$  的一个最大独立集。这两个问题都是 NP-复杂问题。当用算法解决其中一个问题时, 也就解决了另一个问题。例如, 如果有一个求解最大完备子图问题的算法, 则也能解决最大独立集问题, 方法是首先计算所给图的补图, 然后寻找补图的最大完备子图。

例16-10 假定有一个  $n$  个动物构成的集合。可以定义一个有  $n$  个顶点的相容图 (compatibility graph)  $G$ 。当且仅当动物  $u$  和  $v$  相容时,  $(u,v)$  是  $G$  的一条边。 $G$  的一个最大完备子图定义了相互相容的动物构成的最大子集。

15.2节考察了如何找到一个具有最大尺寸的互不交叉的网组的集合问题。可以把这个问题看作是一个最大独立集问题。定义一个图, 图中每个顶点表示一个网组。当且仅当两个顶点对应的网组交叉时, 它们之间有一条边。所以该图的一个最大独立集对应于非交叉网组的一个最大尺寸的子集。当网组有一个端点在路径顶端, 而另一个在底端时, 非交叉网组的最大尺寸的子集能在多项式时间 (实际上是  $\Theta(n^2)$ ) 内用动态规划算法得到。当一个网组的端点可能在平面中的任意地方时, 不可能有在多项式时间内找到非交叉网组的最大尺寸子集的算法。

最大完备子图问题和最大独立集问题可由回溯算法在  $O(n2^n)$  时间内解决。两个问题都可使用子集解空间树 (如图16-2所示)。考察最大完备子图问题, 该递归回溯算法与程序16-3非常类似。当试图移动到空间树的  $i$  层节点  $Z$  的左孩子时, 需要证明从顶点  $i$  到每一个其他的顶点  $j$  ( $x_j=1$  且  $j$  在从根到  $Z$  的路径上) 有一条边。当试图移动到  $Z$  的右孩子时, 需要证明还有足够多的顶点未被搜索, 以便在右子树有可能找到一个较大的完备子图。

回溯算法可作为类 AdjacencyGraph (见程序12-4) 的一个成员来实现, 为此首先要在该类中加入私有静态成员  $x$  (整型数组, 用于存储到当前节点的路径),  $bestx$  (整型数组, 保存目前的最优解),  $bestn$  ( $bestx$  中点的数量),  $cn$  ( $x$  中点的数量)。所以类 AdjacencyGraph 的所有实例都能共享这些变量。

函数  $maxClique$  (见程序16-10) 是类 AdjacencyGraph 的一个私有成员, 而  $MaxClique$  是一个共享成员。函数  $maxClique$  对解空间树进行搜索, 而  $MaxClique$  初始化必要的变量。 $MaxClique(v)$  的执行返回最大完备子图的尺寸, 同时它也设置整型数组  $v$ , 当且仅当顶点  $i$  不是所找到的最大完备子图的一个成员时,  $v[i]=0$ 。

程序16-10 最大完备子图

```
void AdjacencyGraph::maxClique(int i)
```

```

// 计算最大完备子图的回溯代码
if (i > n) { // 在叶子上
    // 找到一个更大的完备子图, 更新
    for (int j = 1; j <= n; j++)
        bestx[j] = x[j];
    bestn = cn;
    return;}

// 在当前完备子图中检查顶点 i 是否与其它顶点相连
int OK = 1;
for (int j = 1; j < i; j++)
    if (x[j] && a[i][j] == NoEdge) {
        // i 不与 j 相连
        OK = 0;
        break;}

if (OK) { // 尝试 x[i] = 1
    x[i] = 1; // 把 i 加入完备子图
    cn++;
    maxClique(i+1);
    x[i] = 0;
    cn--;}

if (cn + n - i > bestn) { // 尝试 x[i] = 0
    x[i] = 0;
    maxClique(i+1);}
}

int AdjacencyGraph::MaxClique(int v[])
// 返回最大完备子图的大小
// 完备子图的顶点放入 v[1:n]
// 初始化
x = new int [n+1];
cn = 0;
bestn = 0;
bestx = v;

// 寻找最大完备子图
maxClique(1);

delete [] x;
return bestn;
}

```

#### 16.2.4 旅行商问题

旅行商问题（例 16.3）的解空间是一个排列树。这样的树可用函数 Perm(见程序 1-10)搜索，并可生成元素表的所有排列。如果以  $x=[1, 2, \dots, n]$  开始，那么通过产生从  $x_2$  到  $x_n$  的所有排列，可生成  $n$  顶点旅行商问题的解空间。由于 Perm 产生具有相同前缀的所有排列，因此

可以容易地改造 Perm，使其不能产生具有不可行前缀（即该前缀没有定义路径）或不可能比当前最优旅行还优的前缀的排列。注意在一个排列空间树中，由任意子树中的叶节点定义的排列有相同的前缀（如图 16-5 所示）。因此，考察时删除特定的前缀等价于搜索期间不进入相应的子树。

旅行商问题的回溯算法可作为类 AdjacencyWDigraph（见程序 12-1）的一个成员。在其他例子中，有两个成员函数：tSP 和 TSP。前者是一个保护或私有成员，后者是一个共享成员。函数 G.TSP(v) 返回最少耗费旅行的花费，旅行自身由整型数组 v 返回。若网络中无旅行，则返回 NoEdge。tSP 在排列空间树中进行递归回溯搜索，TSP 是其一个必要的预处理过程。TSP 假定 x（用来保存到当前节点的路径的整型数组），bestx（保存目前发现的最优旅行的整型数组），cc（类型为 T 的变量，保存当前节点的局部旅行的耗费），bestc（类型为 T 的变量，保存目前最优解的耗费）已被定义为 AdjacencyWDigraph 中的静态数据成员。TSP 见程序 16-11。tSP(2) 搜索一棵包含 x[2:n] 的所有排列的树。

程序 16-11 旅行商回溯算法的预处理程序

```
template<class T>
T AdjacencyWDigraph<T>::TSP(int v[])
{
    // 用回溯算法解决旅行商问题
    // 返回最优旅游路径的耗费，最优路径存入 v[1:n]
    // 初始化
    x = new int [n+1];
    // x 是排列
    for (int i = 1; i <= n; i++)
        x[i] = i;
    bestc = NoEdge;
    bestx = v; // 使用数组 v 来存储最优路径
    cc = 0;

    // 搜索 x[2:n] 的各种排列
    tSP(2);

    delete [] x;
    return bestc;
}
```

函数 tSP 见程序 16-12。它的结构与函数 Perm 相同。当  $i=n$  时，处在排列树的叶节点的父节点上，并且需要验证从  $x[n-1]$  到  $x[n]$  有一条边，从  $x[n]$  到起点  $x[1]$  也有一条边。若两条边都存在，则发现了一个新旅行。在本例中，需要验证是否该旅行是目前发现的最优旅行。若是，则将旅行和它的耗费分别存入 bestx 与 bestc 中。

当  $i < n$  时，检查当前  $i-1$  层节点的孩子节点，并且仅当以下情况出现时，移动到孩子节点之一：1) 有从  $x[i-1]$  到  $x[i]$  的一条边（如果是这样的话， $x[1:i]$  定义了网络中的一条路径）；2) 路径  $x[1:i]$  的耗费小于当前最优解的耗费。变量 cc 保存目前所构造的路径的耗费。

每次找到一个更好的旅行时，除了更新 bestx 的耗费外，tSP 需耗时  $O((n-1)!)$ 。因为需发生  $O((n-1)!)$  次更新且每一次更新的耗费为  $\Theta(n)$  时间，因此更新所需时间为  $O(n*(n-1)!)$ 。通过使用加强的条件（练习 16），能减少由 tSP 搜索的树节点的数量。



程序16-12 旅行商问题的迭代回溯算法

```

void AdjacencyWDigraph<T>::tSP(int i)
{// 旅行商问题的回溯算法
    if (i == n) { // 位于一个叶子的父节点
        // 通过增加两条边来完成旅行
        if (a[x[n-1]][x[n]] != NoEdge &&
            a[x[n]][1] != NoEdge &&
            (cc + a[x[n-1]][x[n]] + a[x[n]][1] < bestc ||
             bestc == NoEdge)) { // 找到更优的旅行路径
            for (int j = 1; j <= n; j++)
                bestx[j] = x[j];
            bestc = cc + a[x[n-1]][x[n]] + a[x[n]][1];
        }
    }
    else { // 尝试子树
        for (int j = i; j <= n; j++)
            // 能移动到子树 x[j] 吗?
            if (a[x[i-1]][x[j]] != NoEdge &&
                (cc + a[x[i-1]][x[j]] < bestc ||
                 bestc == NoEdge)) { // 能
                // 搜索该子树
                Swap(x[i], x[j]);
                cc += a[x[i-1]][x[j]];
                tSP(i+1);
                cc -= a[x[i-1]][x[j]];
                Swap(x[i], x[j]);
            }
    }
}

```

### 16.2.5 电路板排列

在大规模电子系统的设计中存在着电路板排列问题。这个问题的经典形式为将  $n$  个电路板放置到一个机箱的许多插槽中，(如图 16-8 所示)。  $n$  个电路板的每一种排列定义了一种放置方法。令  $B = \{b_1, \dots, b_n\}$  表示这  $n$  个电路板。  $m$  个网组集合  $L = \{N_1, \dots, N_m\}$  由电路板定义，  $N_i$  是  $B$  的子集，子集中的元素需要连接起来。实际中用电线将插有这些电路板的插槽连接起来。

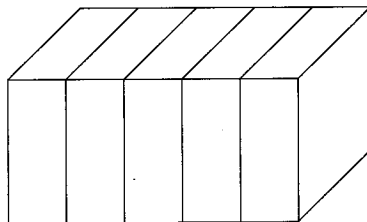


图16-8 电路板及插槽

例16-11 令  $n=8, m=5$ 。集合  $B$  和  $L$  如下：

$$B = \{b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8\}$$

$$L = \{N_1, N_2, N_3, N_4, N_5\}$$

$$N_1 = \{b_4, b_5, b_6\}$$

$$N_2 = \{b_2, b_3\}$$

$$N_3 = \{b_1, b_3\}$$

$$N_4 = \{b_3, b_6\}$$

$$N_5 = \{b_7, b_8\}$$

图16-9给出了电路板的一个可能的排列。边表示在电路板之间的连线。

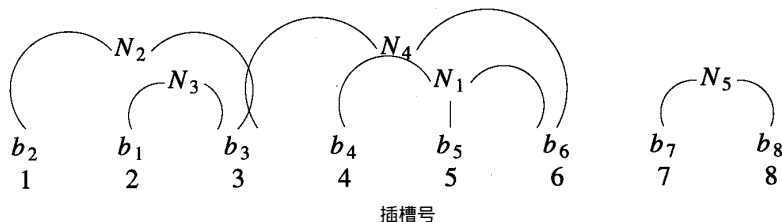


图16-9 电路板布线

令  $x$  为电路板的一个排列。电路板  $x_i$  被放置到机箱的插槽  $i$  中。 $density(x)$  为机箱中任意一对相邻插槽间所连电线数目中的最大值。对于图 16-9 中的排列,  $density$  为 2。有两根电线连接了插槽 2 和 3, 插槽 4 和 5, 插槽 5 和 6。插槽 6 和 7 之间无电线, 余下的相邻插槽都只有一根电线。

板式机箱被设计成具有相同的相邻插槽间距, 因此这个间距决定了机箱的大小。该间距必须保证足够大以便容纳相邻插槽间的连线。因此这个距离 (继而机箱的大小) 由  $density(x)$  决定。

电路板排列问题的目标是找到一种电路板的排列方式, 使其有最小的  $density$ 。既然该问题是一个  $NP$ -复杂问题, 故它不可能由一个多项式时间的算法来解决, 而象回溯这样的搜索方法则是解决该问题的一种较好方法。回溯算法为了找到最优的电路板排列方式, 将搜索一个排列空间。

用一个  $n \times m$  的整型数组  $B$  表示输入, 当且仅当  $N_j$  中包含电路板  $b_i$  时,  $B[i][j]=1$ 。令  $total[j]$  为  $N_j$  中电路板的数量。对于任意部分的电路板排列  $x[1:i]$ , 令  $now[j]$  为既在  $x[1:i]$  中又被包含在  $N_j$  中的电路板的数量。当且仅当  $now[j]>0$  且  $now[j] < total[j]$  时,  $N_j$  在插槽  $i$  和  $i+1$  之间有连线。插槽  $i$  和  $i+1$  间的线密度可利用该测试方法计算出来。在插槽  $k$  和  $k+1$  ( $1 \leq k < i$ ) 间的线密度的最大值给出了局部排列的密度。

为了实现电路板排列问题的回溯算法, 使用了类 `Board` (见程序 16-13)。程序 16-14 给出了私有方法 `BestOrder`, 程序 16-15 给出了函数 `ArrangeBoards`。`ArrangeBoards` 返回最优的电路板排列密度, 最优的排列由数组 `bestx` 返回。

`ArrangeBoards` 创建类 `Board` 的一个成员  $x$  并初始化与之相关的变量。尤其是  $total$  被初始化以使  $total[j]$  等于  $N_j$  中电路板的数量。 $now[1:n]$  被置为 0, 与一个空的局部排列相对应。调用  $x.BestOrder(1,0)$  搜索  $x[1:n]$  的排列树, 以从密度为 0 的空排列中找到一个最优的排列。通常,  $x.BestOrder(i,cd)$  寻找最优的局部排列  $x[1:i-1]$ , 该局部排列密度为  $cd$ 。

函数 `BestOrder` (见程序 16-14) 和程序 16-12 有同样的结构, 它也搜索一个排列空间。当  $i=n$  时, 表示所有的电路板已被放置且  $cd$  为排列的密度。既然这个算法只寻找那些比当前最优排列还优的排列, 所以不必验证  $cd$  是否比  $beste$  要小。当  $i < n$  时, 排列还未被完成。 $x[1:i-1]$  定义了当前节点的局部排列, 而  $cd$  是它的密度。这个节点的每一个孩子通过当前排列的末端增加一个电路板而扩充了这个局部排列。对于每一个这样的扩充, 新的密度  $ld$  被计算, 且只有  $ld < bestd$  的节点被搜索, 其他的节点和它们的子树不被搜索。

在排列树的每一个节点处, 函数 `BestOrder` 花费  $\Theta(m)$  的时间计算每一个孩子节点的密度。所以计算密度的总时间为  $O(mn!)$ 。此外, 产生排列的时间为  $O(n!)$  且更新最优排列的时间为  $O(mn)$ 。

注意每一个更新至少将  $\text{bestd}$  的值减少 1，且最终  $\text{bestd} = 0$ 。所以更新的次数是  $O(m)$ 。BestOrder 的整体复杂性为  $O(mn!)$ 。

程序16-13 Board的类定义

---

```
class Board {
    friend ArrangeBoards(int**, int, int, int []);
private:
    void BestOrder(int i, int cd);
    int *x,          // 到达当前节点的路径
        *bestx,      // 目前的最优排列
        *total,      // total[j] = 带插槽j的板的数目
        *now,        // now[j] = 在含插槽j的部分排列中的板的数目
        bestd,       // bestx的密度
        n,           // 板的数目
        m,           // 插槽的数目
        **B;         // 板的二维数组
};
```

---

程序16-14 搜索排列树

---

```
void Board::BestOrder(int i, int cd)
{
    // 按回溯算法搜索排列树
    if (i == n) {
        for (int j = 1; j <= n; j++)
            bestx[j] = x[j];
        bestd = cd;
    }
    else // 尝试子树
        for (int j = i; j <= n; j++) {
            // 用x[j] 作为下一块电路板对孩子进行尝试
            // 在最后一个插槽更新并计算密度
            int ld = 0;
            for (int k = 1; k <= m; k++) {
                now[k] += B[x[j]][k];
                if (now[k] > 0 && total[k] != now[k])
                    ld++;
            }

            // 更新 ld 为局部排列的总密度
            if (cd > ld) ld = cd;

            // 仅当子树中包含一个更优的排列时，搜索该子树
            if (ld < bestd) {
                // 移动到孩子
                Swap(x[i], x[j]);
                BestOrder(i+1, ld);
                Swap(x[i], x[j]);
            }

            // 重置
            for (k = 1; k <= m; k++)
                now[k] -= B[x[j]][k];
        }
}
```

---

```

    }
}

```

程序16-15 BestOrder(程序16-14)的预处理代码

```

int ArrangeBoards(int **B, int n, int m, int bestx[ ])
{
    // 返回最优密度
    // 在bestx中返回最优排列
    Board X;
    // 初始化X
    X.x = new int [n+1];
    X.total = new int [m+1];
    X.now = new int [m+1];
    X.B = B;
    X.n = n;
    X.m = m;
    X.bestx = bestx;
    X.bestd = m + 1;

    // 初始化total和now
    for (int i = 1; i <= m; i++) {
        X.total[i] = 0;
        X.now[i] = 0;
    }

    // 初始化x并计算 total
    for (i = 1; i <= n; i++) {
        X.x[i] = i;
        for (int j = 1; j <= m; j++)
            X.total[j] += B[i][j];
    }

    // 寻找最优排列
    X.BestOrder(1,0);

    delete [] X.x;
    delete [] X.total;
    delete [] X.now;
    return X.bestd;
}

```

## 练习

4. 证明在两船装载问题中，只要存在一种方法能装载所有货箱，则通过尽可能装满第一艘船就可找到一种可行的装载方法。

5. 运行程序16-3和16-4的代码，测试它们的相对运行时间。

6. 运用16.2.1节中第4小节的方法1) 来更新程序16-3，使其能得到时间复杂性 $O(2^n)$ 。

7. 运用16.2.1节中第4小节的方法2) 来修改程序16-3，使其运行时间减少至 $O(2^n)$ 。

8. 为子集之和问题写一个递归回溯算法。注意，只要找到一个子集，其和为 $c_1$ ，则可终止

程序运行。没有必要记住目前的最优解。代码不应使用像程序 16-3 中的数组  $x_0$ 。在找到和为  $c_i$  的子集之后展开递归，可以重构出最优解。

9. 优化程序 16-7 和 16-9 以便能产生出与背包问题最优解相对应的一个 0/1 数组  $x_0$ 。

10. 用迭代回溯算法求解 0/1 背包问题。该算法与程序 16-4 类似。可以修改 `Knap::Bound`，使其返回被装入背包的最后一个对象  $i$ ，这样可避免根据 `Bound` 重新向左移动而可直接移动到最左节点（原先由 `Bound` 确定）。

11. 编写程序 16-10（与程序 16-4 相对应）的迭代版本并比较这两个版本。

12. 改写程序 16-10，使其首先按度的递减次序来排列各顶点。你认为该版本比程序 16-10 好吗？

13. 编写一个求解最大独立集问题的回溯算法。

14. 重写最大完备子图代码（见程序 16-10），把它作为类 `UNetwork` 的成员。对于类 `AdjacencyGraph`、`AdjacencyWGraph`、`LinkedGraph` 和 `LinkedWGraph`（见 12.7 节）的成员，该代码同样有效。

15. 令  $G$  为一个  $n$  顶点的有向图， $Max_i$  为从顶点  $i$  出发的具有最大耗费的边的耗费

1) 证明旅行商的每一个旅行有一个小于  $\sum_{i=1}^n Max_i + 1$  的耗费。

2) 使用上述界限作为 `bestc` 的初始值。重写 `TSP` 和 `tSP`，尽可能简化它们。

16. 令  $G$  为具有  $n$  个顶点的有向图， $MinOut_i$  为从顶点  $i$  出发的具有最小耗费的边的耗费

1) 证明具有前缀  $x_1$  到  $x_i$  的旅行商的所有旅行耗费至少为  $\sum_{j=2}^i A(x_{j-1}, x_j) + \sum_{y=i}^n MinOut_{x_y}$  其中  $A(u, v)$  是边  $(u, v)$  的耗费。

2) 在程序 16-12 中，使用

```
if (a[x[i-1]][x[j]] != NoEdge &&
    (cc + a[x[i-1]][x[i]] < bestc ||
     bestc == NoEdge))
```

来决定何时移动到一个孩子节点。要求使用 1) 的结果得到一个更强的条件。第一个和可根据 `cc` 计算出来，通过用一个新变量 `r` 保留不在当前路径中的顶点的 `MinOut[i]` 的和，可以很容易地计算出第二个和。

3) 测试 `tSP` 的新版本。与程序 16-12 比较，它访问了排列树的多少节点？

17. 考察电路板排列问题。 $N_i$  的长度为  $N_i$  中第一块和最后一块电路板间的距离。对于图 16-9， $N_4$  中第一个电路板在插槽 3 中，最后一个电路板在插槽 6 中，则  $N_4$  的长度为 3。 $N_2$  的长度为 2。 $N_1$  最大值为 3。编写一个回溯算法以找到具有最小的最大长度的板排列。试测试代码的正确性。

18. [顶点覆盖] 令  $G$  为一个无向图。当且仅当对于  $G$  中的每一条边  $(u, v)$ ， $u$  或  $v$  或  $u, v$  在  $U$  中时， $G$  的顶点子集  $U$  是一个顶点覆盖（vertex cover）。 $U$  中顶点的数量是覆盖的大小。在图 16-7a 中， $\{1, 2, 5\}$  是大小为 3 的一个顶点覆盖。编写一个回溯算法寻找具有最小尺寸的顶点覆盖。算法的复杂性是多少？

19. [简易最大切割] 令  $G$  是一个无向图。 $U$  是  $G$  中顶点的任意子集。 $V$  是  $G$  余下的点的集合。一个端点在  $U$  中，另一个端点在  $V$  中的边的数量是  $U$  所定义的切割（cut）的大小。编写一个回溯算法，寻找最大切割的大小和相应的  $U$ 。算法的复杂性是多少？

20. [机器设计] 某机器由  $n$  个部件组成，每一个部件可从 3 个投资者那里获得。令  $w_{ij}$  是从投资者  $j$  那里得到的零件  $i$  的重量， $c_{ij}$  则为该零件的耗费。编写一个回溯算法，找出耗费不超过  $c$  的机器构成方案，使其重量最少。算法的复杂性是多少？

21. [网络设计] 一个汽油传送网络可由加权的有向无环图  $G$  表示。 $G$  中有一个称为原点的顶点  $S$ 。从  $S$  出发, 汽油被输送到图中的其他顶点。 $S$  的入度为 0, 每一条边上的权给出了它所连接的两点间的距离。通过网络输送汽油时, 压力的损失是所走距离的函数。为了保证网络的正常运转, 在网络传输中必须保证最小压力  $P_{\min}$ 。为了维持这个最小压力, 可将压力放大器放在网络中的一些或全部顶点。压力放大器可将压力恢复至最大可允许的量级  $P_{\max}$ 。令  $d$  为汽油在压力由  $P_{\max}$  降为  $P_{\min}$  时所走的距离。在设置信号放大器问题中, 需要放置最少数量的放大器, 以便在遇到一个放大器之前汽油所走的距离不超过  $d$ 。编写一个回溯算法来求解该问题。算法的复杂性是多少?

22. [ $n$  皇后问题] 在  $n$  皇后问题中, 我们希望在  $n \times n$  的棋盘上找到一个  $n$  皇后的放置方法以便任意两个皇后之间不冲突。当且仅当两个皇后在相同的排、列、对角线或反对角线上时, 她们之间将发生冲突。假定在任何可行的解决方案中, 皇后  $i$  被放置在棋盘的第  $i$  排。所以只对决定每一个皇后所在的列感兴趣。令  $c_i$  为皇后  $i$  所处的列。如果任意两个皇后不冲突, 则  $[c_1, \dots, c_n]$  是  $[1, 2, \dots, n]$  的一个排列。 $n$  皇后问题的解空间因此被限制到  $[1, 2, \dots, n]$  的所有排列中。

1) 将  $n$  皇后的解空间组织成一棵树。

2) 编写一个回溯算法, 搜索  $n$  皇后问题的可行排列。

\*23. 编写一个函数, 使用回溯算法来搜索一个子集空间树, 该树为一个二叉树。函数中的参数应包含如下函数: 确定一个节点是否可行的函数, 计算该节点的界限值的函数, 决定界限是否优于另一个值的函数等。用 0/1 背包问题来测试程序。

\*24. 使用排列空间树来完成练习 23。

\*25. 编写一个函数, 用回溯法搜索一个解空间。函数中的参数应包括下列函数: 产生节点的下一个孩子的函数, 决定下一个孩子是否是可行的函数, 计算该节点界限的函数, 决定该界限值是否优于另一个值的函数等。用 0/1 背包问题来测试程序。

China-pub.com

下载



## 第17章 分枝定界

任何美好的事情都有结束的时候。现在我们学习的是本书的最后一章。幸运的是，本章用到的大部分概念在前面各章中已作了介绍。类似于回溯法，分枝定界法在搜索解空间时，也经常使用树形结构来组织解空间（常用的树结构是第16章所介绍的子集树和排列树）。然而与回溯法不同的是，回溯算法使用深度优先方法搜索树结构，而分枝定界一般用宽度优先或最小耗费方法来搜索这些树。本章与第16章所考察的应用完全相同，因此，可以很容易比较回溯法与分枝定界法的异同。

相对而言，分枝定界算法的解空间比回溯法大得多，因此当内存容量有限时，回溯法成功的可能性更大。

### 17.1 算法思想

分枝定界（branch and bound）是另一种系统地搜索解空间的方法，它与回溯法的主要区别在于对E-节点的扩充方式。每个活节点有且仅有一次机会变成E-节点。当一个节点变为E-节点时，则生成从该节点移动一步即可到达的所有新节点。在生成的节点中，抛弃那些不可能导出（最优）可行解的节点，其余节点加入活节点表，然后从表中选择一个节点作为下一个E-节点。从活节点表中取出所选择的节点并进行扩充，直到找到解或活动表为空，扩充过程才结束。

有两种常用的方法可用来选择下一个E-节点（虽然也可能存在其他的方法）：

1) 先进先出（FIFO） 即从活节点表中取出节点的顺序与加入节点的顺序相同，因此活节点表的性质与队列相同。

2) 最小耗费或最大收益法 在这种模式中，每个节点都有一个对应的耗费或收益。如果查找一个具有最小耗费的解，则活节点表可用最小堆来建立，下一个E-节点就是具有最小耗费的活节点；如果希望搜索一个具有最大收益的解，则可用最大堆来构造活节点表，下一个E-节点是具有最大收益的活节点。

例17-1 [迷宫老鼠] 考察图16-3a 给出的迷宫老鼠例子和图16-1的解空间结构。使用FIFO分枝定界，初始时取（1，1）作为E-节点且活动队列为空。迷宫的位置（1,1）被置为1，以免再次返回到这个位置。（1，1）被扩充，它的相邻节点（1，2）和（2，1）加入到队列中（即活节点表）。为避免再次回到这两个位置，将位置（1，2）和（2，1）置为1。此时迷宫如图17-1a所示，E-节点（1，1）被删除。

1 1 0  
1 1 1  
0 0 0  
a)

1 1 1  
1 1 1  
0 0 0  
b)

1 1 1  
1 1 1  
1 0 0  
c)

图17-1 迷宫问题的FIFO分枝定界方法

节点  $(1, 2)$  从队列中移出并被扩充。检查它的三个相邻节点 (见图 16-1 的解空间), 只有  $(1, 3)$  是可行的移动 (剩余的两个节点是障碍节点), 将其加入队列, 并把相应的迷宫位置置为 1, 所得到的迷宫状态如图 17-1b 所示。节点  $(1, 2)$  被删除, 而下一个 E-节点  $(2, 1)$  将会被取出, 当此节点被展开时, 节点  $(3, 1)$  被加入队列中, 节点  $(3, 1)$  被置为 1, 节点  $(2, 1)$  被删除, 所得到的迷宫如图 17-1c 所示。此时队列中包含  $(1, 3)$  和  $(3, 1)$  两个节点。随后节点  $(1, 3)$  变成下一个 E-节点, 由于此节点不能到达任何新的节点, 所以此节点即被删除, 节点  $(3, 1)$  成为新的 E-节点, 将队列清空。节点  $(3, 1)$  展开,  $(3, 2)$  被加入队列中, 而  $(3, 1)$  被删除。 $(3, 2)$  变为新的 E-节点, 展开此节点后, 到达节点  $(3, 3)$ , 即迷宫的出口。

使用 FIFO 搜索, 总能找出从迷宫入口到出口的最短路径。需要注意的是: 利用回溯法找到的路径却不一定是最短路径。有趣的是, 程序 6-11 已经给出了利用 FIFO 分枝定界搜索从迷宫的  $(1, 1)$  位置到  $(n, n)$  位置的最短路径的代码。

**例 17-2 [0/1 背包问题]** 下面比较分别利用 FIFO 分枝定界和最大收益分枝定界方法来解决如下背包问题:  $n=3$ ,  $w=[20, 15, 15]$ ,  $p=[40, 25, 25]$ ,  $c=30$ 。FIFO 分枝定界利用一个队列来记录活节点, 节点将按照 FIFO 顺序从队列中取出; 而最大收益分枝定界使用一个最大堆, 其中的 E-节点按照每个活节点收益值的降序, 或是按照活节点任意子树的叶节点所能获得的收益估计值的降序从队列中取出。本例所使用的背包问题与例 16.2 相同, 并且有相同的解空间树。

使用 FIFO 分枝定界法搜索, 初始时以根节点 A 作为 E-节点, 此时活节点队列为空。当节点 A 展开时, 生成了节点 B 和 C, 由于这两个节点都是可行的, 因此都被加入活节点队列中, 节点 A 被删除。下一个 E-节点是 B, 展开它并产生了节点 D 和 E, D 是不可行的, 被删除, 而 E 被加入队列中。下一步节点 C 成为 E-节点, 它展开后生成节点 F 和 G, 两者都是可行节点, 加入队列中。下一个 E-节点 E 生成节点 J 和 K, J 不可行而被删除, K 是一个可行的叶节点, 并产生一个到目前为止可行的解, 它的收益值为 40。

下一个 E-节点是 F, 它产生两个孩子 L、M, L 代表一个可行的解且其收益值为 50, M 代表另一个收益值为 15 的可行解。G 是最后一个 E-节点, 它的孩子 N 和 O 都是可行的。由于活节点队列变为空, 因此搜索过程终止, 最佳解的收益值为 50。

可以看到, 工作在解空间树上的 FIFO 分枝定界方法非常象从根节点出发的宽度优先搜索。它们的主要区别是在 FIFO 分枝定界中不可行的节点不会被搜索。

最大收益分枝定界算法以解空间树中的节点 A 作为初始节点。展开初始节点得到节点 B 和 C, 两者都是可行的并被插入堆中, 节点 B 获得的收益值是 40 (设  $x_1=1$ ), 而节点 C 得到的收益值为 0。A 被删除, B 成为下一个 E-节点, 因为它的收益值比 C 的大。当展开 B 时得到了节点 D 和 E, D 是不可行的而被删除, E 加入堆中。由于 E 具有收益值 40, 而 C 为 0, 因为 E 成为下一个 E-节点。展开 E 时生成节点 J 和 K, J 不可行而被删除, K 是一个可行的解, 因此 K 作为目前能找到的最优解而记录下来, 然后 K 被删除。由于只剩下一个活节点 C 在堆中, 因此 C 作为 E-节点被展开, 生成 F、G 两个节点插入堆中。F 的收益值为 25, 因此成为下一个 E-节点, 展开后得到节点 L 和 M, 但 L、M 都被删除, 因为它们是叶节点, 同时 L 所对应的解被作为当前最优解记录下来。最终, G 成为 E-节点, 生成的节点为 N 和 O, 两者都是叶节点而被删除, 两者所对应的解都不比当前的最优解更好, 因此最优解保持不变。此时堆变为空, 没有下一个 E-节点产生, 搜索过程终止。终止于 J 的搜索即为最优解。

犹如在回溯方法中一样, 可利用一个定界函数来加速最优解的搜索过程。定界函数为最大收益设置了一个上限, 通过展开一个特殊的节点可能获得这个最大收益。如果一个节点的定界

函数值不大于目前最优解的收益值,则此节点会被删除而不作展开,更进一步,在最大收益分枝定界方法中,可以使节点按照它们收益的定界函数值的非升序从堆中取出,而不是按照节点的实际收益值来取出。这种策略从可能到达一个好的叶节点的活节点出发,而不是从目前具有较大收益值的节点出发。

例17-3 [旅行商问题] 对于图16-4的四城市旅行商问题,其对应的解空间为图16-5所示的排列树。FIFO分枝定界使用节点B作为初始的E-节点,活节点队列初始为空。当B展开时,生成节点C、D和E。由于从顶点1到顶点2,3,4都有边相连,所以C、D、E三个节点都是可行的并加入队列中。当前的E-节点B被删除,新的E-节点是队列中的第一个节点,即节点C。因为在图16-4中存在从顶点2到顶点3和4的边,因此展开C,生成节点F和G,两者都被加入队列。下一步,D成为E-节点,接着又是E,到目前为止活节点队列中包含节点F到K。

下一个E-节点是F,展开它得到了叶节点L。至此找到了一个旅行路径,它的开销是59。展开下一个E-节点G,得到叶节点M,它对应于一个开销为66的旅行路径。接着H成为E-节点,从而找到叶节点N,对应开销为25的旅行路径。下一个E-节点是I,它对应的部分旅行1-3-4的开销已经为26,超过了目前最优的旅行路径,因此,I不会被展开。最后,节点J,K成为E-节点并被展开。经过这些展开过程,队列变为空,算法结束。找到的最优方案是节点N所对应的旅行路径。

如果不使用FIFO方法,还可以使用最小耗费方法来搜索解空间树,即用一个最小堆来存储活节点。这种方法同样从节点B开始搜索,并使用一个空的活节点列表。当节点B展开时,生成节点C、D和E并将它们加入最小堆中。在最小堆的节点中,E具有最小耗费(因为1-4的局部旅行的耗费是4),因此成为E-节点。展开E生成节点J和K并将它们加入最小堆,这两个节点的耗费分别为14和24。此时,在所有最小堆的节点中,D具有最小耗费,因而成为E-节点,并生成节点H和I。至此,最小堆中包含节点C、H、I、J和K,H具有最小耗费,因此H成为下一个E-节点。展开节点E,得到一个完整的旅行路径1-3-2-4-1,它的开销是25。节点J是下一个E-节点,展开它得到节点P,它对应于一个耗费为25的旅行路径。节点K和I是下两个E-节点。由于I的开销超过了当前最优的旅行路径,因此搜索结束,而剩下的所有活节点都不能使我们找到更优的解。

对于例17-2的背包问题,可以使用一个定界函数来减少生成和展开的节点数量。这种函数将确定旅行的最小耗费的下限,这个下限可通过展开某个特定的节点而得到。如果一个节点的定界函数值不能比当前的最优旅行更小,则它将被删除而不被展开。另外,对于最小耗费分枝定界,节点按照它在最小堆中的非降序取出。

在以上几个例子中,可以利用定界函数来降低所产生的树型解空间的节点数目。当设计定界函数时,必须记住主要目的是利用最少的时间,在内存允许的范围内去解决问题。而通过产生具有最少节点的树来解决问题并不是根本的目标。因此,我们需要的是一个能够有效地减少计算时间并因此而使产生的节点数目也减少的定界函数。

回溯法比分枝定界在占用内存方面具有优势。回溯法占用的内存是 $O(\text{解空间的} \text{最大路径长度})$ ,而分枝定界所占用的内存为 $O(\text{解空间大小})$ 。对于一个子集空间,回溯法需要 $\Theta(n)$ 的内存空间,而分枝定界则需要 $O(2^n)$ 的空间。对于排列空间,回溯需要 $\Theta(n)$ 的内存空间,分枝定界需要 $O(n!)$ 的空间。虽然最大收益(或最小耗费)分枝定界在直觉上要优于回溯法,并且在许多情况下可能会比回溯法检查更少的节点,但在实际应用中,它可能会在回溯法超出允许的时间限制之前就超出了内存的限制。

## 练习

1. 假定在一个LIFO分枝定界搜索中，活节点列表的行为与堆栈相同，请使用这种方法来解决例17-2的背包问题。LIFO分枝定界与回溯有何区别？

2. 对于如下0/1背包问题： $n=4$ ,  $p=[4,3,2,1]$ ,  $w=[1,2,3,4]$ ,  $c=6$ 。

1) 画出有四个对象的背包问题的解空间树。

2) 像例17-2那样，描述用FIFO分枝定界法解决上述问题的过程。

3) 使用程序16-6的Bound函数来计算子树上任一叶节点可能获得的最大收益值，并根据每一步所能得到的最优解对应的定界函数值来判断是否将节点加入活节点列表中。解空间中哪些节点是使用以上机制的FIFO分枝定界方法产生的？

4) 像例17-2那样，描述用最大收益分枝定界法解决上述问题的过程。

5) 在最大收益分枝定界中，若使用3)中的定界函数，将产生解空间树中的哪些节点？

## 17.2 应用

## 17.2.1 货箱装船

## 1. FIFO分枝定界

16.2.1节的货箱装船问题主要是寻找第一条船的最大装载方案。这个问题是一个子集选择问题，它的解空间被组织成一个子集树。对程序16-1进行改造，即得到程序17-1中的FIFO分枝定界代码。程序17-1只是寻找最大装载的重量。

程序17-1 货箱装船问题的FIFO分枝定界算法

```
template<class T>
void AddLiveNode(LinkedQueue<T> &Q, T wt,
                T& bestw, int i, int n)
// 如果不是叶节点，则将节点权值 wt加入队列Q
if (i == n) { // 叶子
    if (wt > bestw) bestw = wt;
} else Q.Add(wt); // 不是叶子
}
```

```
template<class T>
T MaxLoading(T w[], T c, int n)
// 返回最优装载值
// 使用FIFO分枝定界算法
// 为层次1 初始化
LinkedQueue<T> Q; // 活节点队列
Q.Add(-1); // 标记本层的尾部
int i = 1; // E - 节点的层
T Ew = 0, // E - 节点的权值
bestw = 0; // 目前的最优值
```

```
// 搜索子集空间树
while (true) {
```

```

// 检查E-节点的左孩子
if (Ew + w[i] <= c) // x[i] = 1
    AddLiveNode(Q, Ew + w[i], bestw, i, n);

// 右孩子总是可行的
AddLiveNode(Q, Ew, bestw, i, n); // x[i] = 0

Q.Delete(Ew); // 取下一个E-节点
if (Ew == -1) { // 到达层的尾部
    if (Q.IsEmpty()) return bestw;
    Q.Add(-1); // 添加尾部标记
    Q.Delete(Ew); // 取下一个E-节点
    i++; // Ew的层
}
}
}

```

其中函数MaxLoading在解空间树中进行分枝定界搜索。链表队列Q用于保存活节点，其中记录着各活节点对应的权值。队列还记录了权值-1，以标识每一层的活节点的结尾。函数AddLiveNode用于增加节点（即把节点对应的权值加入活节点队列），该函数首先检验 $i$ （当前E-节点在解空间树中的层）是否等于 $n$ ，如果相等，则已到达了叶节点。叶节点不被加入队列中，因为它们不被展开。搜索中所到达的每个叶节点都对应着一个可行的解，而每个解都会与目前的最优解来比较，以确定最优解。如果 $i < n$ ，则节点 $i$ 就会被加入队列中。

MaxLoading函数首先初始化 $i=1$ （因为当前E-节点是根节点）， $bestw=0$ （目前最优解的对应值），此时，活节点队列为空。下一步，-1被加入队列以说明正处在第一层的末尾。当前E-节点对应的权值为 $E_w$ 。在while循环中，首先检查节点的左孩子是否可行。如果可行，则调用AddLiveNode，然后将右孩子加入队列（此节点必定是可行的），注意到AddLiveNode可能会失败，因为可能没有足够的内存来给队列增加节点。AddLiveNode并没有去捕获Q.Add中的NoMem异常，这项工作留给用户完成。

如果E-节点的两个孩子都已经被生成，则删除该E-节点。从队列中取出下一个E-节点，此时队列必不为空，因为队列中至少含有本层末尾的标识-1。如果到达了某一层的结尾，则从下一层寻找活节点，当且仅当队列不为空时这些节点存在。当下一层存在活节点时，向队列中加入下一层的结尾标志并开始处理下一层的活节点。

MaxLoading函数的时间和空间复杂性都是 $O(2^n)$ 。

## 2. 改进

我们可以尝试使用程序16-2的优化方法改进上述问题的求解过程。在程序16-2中，只有当右孩子对应的重量加上剩余货箱的重量超出 $bestw$ 时，才选择右孩子。而在程序17-1中，在 $i$ 变为 $n$ 之前， $bestw$ 的值一直保持不变，因此在 $i$ 等于 $n$ 之前对右孩子的测试总能成功，因为 $bestw=0$ 且 $r>0$ 。当 $i$ 等于 $n$ 时，不会再有节点加入队列中，因此这时对右孩子的测试不再有效。

如想要使右孩子的测试仍然有效，应当提早改变 $bestw$ 的值。我们知道，最优装载的重量是子集树中可行节点的重量的最大值。由于仅在向左子树移动时这些重量才会增大，因此可以在每次进行这种移动时改变 $bestw$ 的值。根据以上思想，我们设计了程序17-2。当活节点加入队列时， $w_i$ 不会超过 $bestw$ ，故 $bestw$ 不用更新。因此用一条直接插入MaxLoading的简单语句取代了函数AddLiveNode。

程序17-2 对程序17-1改进之后

```

template<class T>
T MaxLoading(T w[], T c, int n)
{// 返回最优装载值
// 使用FIFO分枝定界算法
// 为层1初始化
LinkedQueue<T> Q;    // 活节点队列
Q.Add(-1);           // 标记本层的尾部
int i = 1;           // E - 节点的层
T Ew = 0,             // E - 节点的重量
bestw = 0;           // 目前的最优值
r = 0;               // E - 节点中余下的重量
for (int j = 2; j <= n; j++)
    r += w[j];

// 搜索子集空间树
while (true) {
    // 检查E - 节点的左孩子
    T wt = Ew + w[i]; // 左孩子的权值
    if (wt <= c) {     // 可行的左孩子
        if (wt > bestw) bestw = wt;
        // 若不是叶子，则添加到队列中
        if (i < n) Q.Add(wt);}

    // 检查右孩子
    if (Ew + r > bestw && i < n)
        Q.Add(Ew); // 可以有一个更好的叶子

    Q.Delete(Ew);    // 取下一个 E - 节点
    if (Ew == -1) {  // 到达层的尾部
        if (Q.IsEmpty()) return bestw;
        Q.Add(-1);   // 添加尾部标记
        Q.Delete(Ew); // 取下一个 E - 节点
        i++;         // E - 节点的层
        r -= w[i];    // E - 节点中余下的重量
    }
}
}

```

### 3. 寻找最优子集

为了找到最优子集，需要记录从每个活节点到达根的路径，因此在找到最优装载所对应的叶节点之后，就可以利用所记录的路径返回到根节点来设置  $x$  的值。活节点队列中元素的类型是 `QNode` (见程序17-3)。这里，当且仅当节点是它的父节点的左孩子时，`LChild` 为 `true`。

程序17-3 类 `QNode`

```

template<class T>
class QNode {
private:

```



```

QNode *parent; // 父节点指针
bool LChild; // 当且仅当是父节点的左孩子时，取值为 true
T weight; // 由到达本节点的路径所定义的部分解的值
};

```

程序17-4是新的分枝定界方法的代码。为了避免使用大量的参数来调用 AddLiveNode，可以把该函数定义为一个内部函数。使用内部函数会使空间需求稍有增加。此外，还可以把 AddLiveNode和MaxLoading定义成类成员函数，这样，它们就可以共享诸如 Q,i,n,bestw,E,bestE和bestw 等类成员。

程序17-4并未删除类型为 QNode的节点。为了删除这些节点，可以保存由 AddLiveNode创建的所有节点的指针，以便在程序结束时删除这些节点。

程序17-4 计算最优子集的分枝定界算法

```

template<class T>
void AddLiveNode(LinkedQueue<QNode<T>*> &Q, T wt, int i, int n, T bestw, QNode<T> *E,
    QNode<T> *&bestE, int bestx[], bool ch)
// 如果不是叶节点，则向队列 Q中添加一个 i 层、重量为 wt的活节点
// 新节点是 E 的一个孩子。当且仅当新节点是左孩子时，ch为true。
// 若是叶子，则 ch取值为 bestx[n]
if (i == n) { // 叶子
    if (wt == bestw) {
        // 目前的最优解
        bestE = E;
        bestx[n] = ch;}
    return;}

// 不是叶子，添加到队列中
QNode<T> *b;
b = new QNode<T>;
b->weight = wt;
b->parent = E;
b->LChild = ch;
Q.Add(b);
}

template<class T>
T MaxLoading(T w[], T c, int n, int bestx[])
// 返回最优装载值，并在 bestx中返回最优装载
// 使用FIFO分枝定界算法
// 初始化层 1
LinkedQueue<QNode<T>*> Q; // 活节点队列
Q.Add(0); // 0 代表本层的尾部
int i = 1; // E - 节点的层
T Ew = 0, // E - 节点的重量
    bestw = 0; // 迄今得到的最优值
r = 0; // E - 节点中余下的重量
for (int j = 2; j <= n; j++)

```



```

    r += w[i];
    QNode<T> *E = 0,           // 当前的 E - 节点
    *bestE;                   // 目前最优的 E - 节点

// 搜索子集空间树
while (true) {
    // 检查E - 节点的左孩子
    T wt = Ew + w[i];
    if (wt <= c) { // 可行的左孩子
        if (wt > bestw) bestw = wt;
        AddLiveNode(Q, wt, i, n, bestw, E, bestE, bestx, true);}

    // 检查右孩子
    if (Ew + r > bestw) AddLiveNode(Q, Ew, i, n, bestw, E, bestE, bestx, false);

    Q.Delete(E);           // 下一个E - 节点
    if (!E) {              // 层的尾部
        if (Q.IsEmpty()) break;
        Q.Add(0);          // 层尾指针
        Q.Delete(E);       // 下一个E - 节点
        i++;               // E - 节点的层次
        r -= w[i];         // E - 节点中余下的重量

    Ew = E ->weight;       // 新的E - 节点的重重量
    }

// 沿着从bestE到根的路径构造x[] , x[n]由 AddLiveNode来设置
for (j = n - 1; j > 0; j--) {
    bestx[j] = bestE ->LChild; // 从bool转换为int
    bestE = bestE ->parent;
}

return bestw;
}

```

#### 4. 最大收益分枝定界

在对子集树进行最大收益分枝定界搜索时，活节点列表是一个最大优先级队列，其中每个活节点 $x$ 都有一个相应的重量上限（最大收益）。这个重量上限是节点 $x$ 相应的重量加上剩余货箱的总重量，所有的活节点按其重量上限的递减顺序变为 $E$ -节点。需要注意的是，如果节点 $x$ 的重量上限是 $x.uweight$ ，则在子树中不可能存在重量超过 $x.uweight$ 的节点。另外，当叶节点对应的重量等于它的重量上限时，可以得出结论：在最大收益分枝定界算法中，当某个叶节点成为 $E$ -节点并且其他任何活节点都不会帮助我们找到具有更大重量的叶节点时，最优装载的搜索终止。

上述策略可以用两种方法来实现。在第一种方法中，最大优先级队列中的活节点都是互相独立的，因此每个活节点内部必须记录从子集树的根到此节点的路径。一旦找到了最优装载所对应的叶节点，就利用这些路径信息来计算 $x$ 值。在第二种方法中，除了把节点加入最大优先

队列之外，节点还必须放在另一个独立的树结构中，这个树结构用来表示所生成的子集树的一部分。当找到最大装载之后，就可以沿着路径从叶节点一步一步返回到根，从而计算出  $x$  值。本书使用第二种方法，第一种方法的实现留作习题。

最大优先队列可用 HeapNode 类型的最大堆来表示（见程序 17-5）。uweight 是活节点的重量上限，level 是活节点所在子集树的层，ptr 是指向活节点在子集树中位置的指针。子集树中节点的类型是 bbnode（见程序 17-5）。节点按 uweight 值从最大堆中取出。

程序17-5 bbnode 和HeapNode 类

---

```
class bbnode {
private:
    bbnode *parent; // 父节点指针
    bool LChild; // 当且仅当是父节点的左孩子时，取值为 true
};

template<class T>
class HeapNode {
public:
    operator T () const {return uweight;}
private:
    bbnode *ptr; // 活节点指针
    T uweight; // 活节点的重量上限
    int level; // 活节点所在层
};
```

---

程序17-6中的函数AddLiveNode用于把bbnode类型的活节点加到子树中，并把HeapNode类型的活节点插入最大堆。AddLiveNode必须被定义为bbnode和HeapNode的友元。

程序17-6

---

```
template<class T>
void AddLiveNode(MaxHeap<HeapNode<T> > &H, bbnode *E, T wt, bool ch, int lev)
// 向最大堆H中增添一个层为lev上限重量为wt的活节点
// 新节点是 E的一个孩子
// 当且仅当新节点是左孩子 ch 为true
{
    bbnode *b = new bbnode;
    b->parent = E;
    b->LChild = ch;
    HeapNode<T> N;
    N.uweight = wt;
    N.level = lev;
    N.ptr = b;
    H.Insert(N);
}

template<class T>
T MaxLoading(T w[], T c, int n, int bestx[])
// 返回最优装载值，最优装载方案保存于 bestx
```

```

// 使用最大收益分枝定界算法
// 定义一个最多有1000个活节点的最大堆
MaxHeap<HeapNode<T>> H(1000);

// 第一剩余重量的数组
// r[j] 为 w[j+1:n]的重量之和
T *r = new T [n+1];
r[n] = 0;
for (int j = n-1; j > 0; j--)
    r[j] = r[j+1] + w[j+1];

// 初始化层1
int i = 1;           // E - 节点的层
bbnode *E = 0;       // 当前E - 节点
T Ew = 0;            // E - 节点的重量

// 搜索子集空间树
while (i != n+1) { // 不在叶子上
    // 检查E - 节点的孩子
    if (Ew + w[i] <= c) { // 可行的左孩子
        AddLiveNode(H, E, Ew+w[i]+r[i], true, i+1);}
    // 右孩子
    AddLiveNode(H, E, Ew+r[i], false, i+1);

    // 取下一个E - 节点
    HeapNode<T> N;
    H.DeleteMax(N); // 不能为空
    i = N.level;
    E = N.ptr;
    Ew = N.uweight - r[i-1];
}

// 沿着从E - 节点E到根的路径构造bestx[]
for (int j = n; j > 0; j--) {
    bestx[j] = E ->LChild; // 从bool转换为int
    E = E ->parent;
}

return Ew;
}

```

函数MaxLoading（见程序17-6）首先定义了一个容量为1000的最大堆，因此，可以用它来解决优先队列中活节点数在任何时候都不超过1000的装箱问题。对于更大型的问题，需要一个容量更大的最大堆。接着，函数MaxLoading初始化剩余重量数组r。第i+1层的节点（即x[1:i]的值都已确定）对应的剩余容器总重量可以用如下公式求出： $r[i] = \sum_{j=i+1}^n w[j]$ 。变量E指向子集树中的当前E-节点，Ew是该节点对应的重量，i是它所在的层。初始时，根节点是E-节点，因此取i=1, Ew=0。由于没有明确地存储根节点，因此E的初始值取为0。

while 循环用于产生当前E-节点的左、右孩子。如果左孩子是可行的（即它的重量没有超

出容量), 则将它加入到子集树中并作为一个第  $i+1$  层节点加入最大堆中。一个可行的节点的右孩子也被认为是可行的, 它总被加入子树及最大堆中。在完成添加操作后, 接着从最大堆中取出下一个 E- 节点。如果没有下一个 E- 节点, 则不存在可行的解。如果下一个 E- 节点是叶节点 (即是一个层为  $n+1$  的节点), 则它代表着一个最优的装载, 可以沿着从叶到根的路径来确定装载方案。

### 5. 说明

1) 使用最大堆来表示活节点的最大优先队列时, 需要预测这个队列的最大长度 (程序 17-6 中是 1000)。为了避免这种预测, 可以使用一个基于指针的最大优先队列来取代基于数组的队列, 这种表示方法见 9.4 节的左高树。

2)  $bestw$  表示当前所有可行节点的重量的最大值, 而优先队列中可能有许多其  $uweight$  不超过  $bestw$  的活节点, 因此这些节点不可能帮助我们找到最优的叶节点, 这些节点浪费了珍贵的队列空间, 并且它们的插入/删除动作也浪费了时间, 所以可以将这些节点删除。有一种策略可以减少这种浪费, 即在插入某个节点之前检查是否有  $uweight < bestw$ 。然而, 由于  $bestw$  在算法执行过程中是不断增大的, 所以目前插入的节点在以后并不能保证  $uweight < bestw$ 。另一种更好的方法是在每次  $bestw$  增大时, 删除队列中所有  $uweight < bestw$  的节点。这种策略要求删除具有最小  $uweight$  的节点。因此, 队列必须支持如下的操作: 插入、删除最大节点、删除最小节点。这种优先队列也被称作双端优先队列 (double-ended priority queue)。这种队列的数据结构描述见第 9 章的参考文献。

## 17.2.2 0/1 背包问题

0/1 背包问题的最大收益分枝定界算法可以由程序 16-6 发展而来。可以使用程序 16-6 的 Bound 函数来计算活节点 N 的收益上限  $up$ , 使得以 N 为根的子树中的任一节点的收益值都不可能超过  $up$ 。活节点的最大堆使用  $up$  作为关键值域, 最大堆的每个入口都以 HeapNode 作为其类型, HeapNode 有如下私有成员:  $up$ , profit, weight, level, ptr, 其中 level 和 ptr 的定义与装箱问题 (见程序 17-5) 中的含义相同。对任一节点 N, N.profit 是 N 的收益值, N.up 是它的收益上限, N.weight 是它对应的重量。bbnode 类型如程序 17-5 中的定义, 各节点按其  $up$  值从最大堆中取出。

程序 17-7 使用了类 Knap, 它类似于回溯法中的类 Knap (见程序 16-5)。两个 Knap 版本中数据成员之间的区别见程序 17-7: 1)  $bestp$  不再是一个成员; 2)  $bestx$  是一个指向 int 的新成员。新增成员的作用是: 当且仅当物品 j 包含在最优解中时,  $bestx[j]=1$ 。函数 AddLiveNode 用于将新的 bbnode 类型的活节点插入子集树中, 同时将 HeapNode 类型的活节点插入到最大堆中。这个函数与装箱问题 (见程序 17-6) 中的对应函数非常类似, 因此相应的代码被省略。

程序 17-7 0/1 背包问题的最大收益分枝定界算法

```
template<class Tw, class Tp>
Tp Knap<Tw, Tp>::MaxProfitKnapsack()
// 返回背包最优装载的收益
//  $bestx[i] = 1$  当且仅当物品 i 属于最优装载
// 使用最大收益分枝定界算法
// 定义一个最多可容纳 1000 个活节点的最大堆
H = new MaxHeap<HeapNode<Tp, Tw>> (1000);

// 为  $bestx$  分配空间
```

```

bestx = new int [n+1];

// 初始化层 1
int i = 1;
E = 0;
cw = cp = 0;
Tp bestp = 0;    // 目前的最优收益
Tp up = Bound(1); // 在根为E的子树中最大可能的收益

// 搜索子集空间树
while (i != n+1) { // 不是叶子
    // 检查左孩子
    Tw wt = cw + w[i];
    if (wt <= c) { // 可行的左孩子
        if (cp+p[i] > bestp) bestp = cp+p[i];
        AddLiveNode(up, cp+p[i], cw+w[i], true, i+1);
        up = Bound(i+1);

        // 检查右孩子
        if (up >= bestp) // 右孩子有希望
            AddLiveNode(up, cp, cw, false, i+1);

        // 取下一个E-节点
        HeapNode<Tp, Tw> N;
        H->DeleteMax(N); // 不能为空
        E = N.ptr;
        cw = N.weight;
        cp = N.profit;
        up = N.upprofit;
        i = N.level;
    }

    // 沿着从E-节点E到根的路径构造bestx[]
    for (int j = n; j > 0; j--) {
        bestx[j] = E->LChild;
        E = E->parent;
    }

    return cp;
}

```

函数MaxProfitKnapsack在子集树中执行最大收益分枝定界搜索。函数假定所有的物品都是按收益密度值的顺序排列，可以使用类似于程序 16-9中回溯算法所使用的预处理代码来完成这种排序。函数MaxProfitKnapsack首先初始化活节点的最大堆，并使用一个数组 bestx来记录最优解。由于需要不断地利用收益密度来排序，物品的索引值会随之变化，因此必须将MaxProfitKnapsack所生成的结果映射回初始时的物品索引。可以用 Q的ID域来实现上述映射（见程序 16-9）。

在函数MaxProfitKnapSack中，E是当前E-节点，cw是节点对应的重量，cp是收益值，up是以E为根的子树中任一节点的收益值上限。while循环一直执行到一个叶节点成为E-节点为止。

由于最大堆中的任何剩余节点都不可能具有超过当前叶节点的收益值，因此当前叶即对应了一个最优解。可以从叶返回到根来确定这个最优解。

MaxProfitKnapsack中while循环的结构很类似于程序17-6的while循环。首先，检验E-节点左孩子的可行性，如它是可行的，则将它加入子集树及活节点队列（即最大堆）；仅当节点右孩子的Bound值指明有可能找到一个最优解时才将右孩子加入子集树和队列中。

### 17.2.3 最大完备子图

16.2.3节完备子图问题的解空间树也是一个子集树，故可以使用与装箱问题、背包问题相同的最大收益分枝定界方法来求解这种问题。解空间树中的节点类型为bbnode，而最大优先队列中元素的类型则是CliqueNode。CliqueNode有如下域：cn（该节点对应的完备子图中的顶点数目），un（该节点的子树中任意叶节点所对应的完备子图的最大尺寸），level（节点在解空间树中的层），cn（当且仅当该节点是其父节点的左孩子时，cn为1），ptr（指向节点在解空间树中的位置）。un的值等于cn+n-level+1。因为根据un和cn（或level）可以求出level（或cn），所以可以去掉cn或level域。当从最大优先队列中选取元素时，选取的是具有最大un值的元素。在程序17-8中，CliqueNode包含了所有的三个域：cn，un和level，这样便于尝试为un赋予不同的含义。函数AddCliqueNode用于向生成的子树和最大堆中加入节点，由于其代码非常类似于装箱和背包问题中的对应函数，故将它略去。

函数BBMaxClique在解空间树中执行最大收益分枝定界搜索，树的根作为初始的E-节点，该节点并没有在所构造的树中明确存储。对于这个节点来说，其cn值（E-节点对应的完备子图的大小）为0，因为还没有任何顶点被加入完备子图中。E-节点的层由变量i指示，它的初值为1，对应于树的根节点。当前所找到的最大完备子图的大小保存在bestn中。

在while循环中，不断展开E-节点直到一个叶节点变成E-节点。对于叶节点， $un = cn$ 。由于所有其他节点的un值都小于等于当前叶节点对应的un值，所以它们不可能产生更大的完备子图，因此最大完备子图已经找到。沿着生成的树中从叶节点到根的路径，即可构造出这个最大完备子图。

为了展开一个非叶E-节点，应首先检查它的左孩子，如果左孩子对应的顶点i与当前E-节点所包含的所有顶点之间都有一条边，则i被加入当前的完备子图之中。为了检查左孩子的可行性，可以沿着从E-节点到根的路径，判断哪些顶点包含在E-节点之中，同时检查这些顶点中每个顶点是否都存在一条到i的边。如果左孩子是可行的，则把它加入到最大优先队列和正在构造的树中。下一步，如果右孩子的子树中包含最大完备子图对应的叶节点，则把右孩子也加入。

由于每个图都有一个最大完备子图，因此从堆中删除节点时，不需要检验堆是否为空。仅当到达一个可行的叶节点时，while循环终止。

程序17-8 最大完备子图问题的分枝定界算法

```
int AdjacencyGraph::BBMaxClique(int bestx[])
// 寻找一个最大完备子图的最大收益分枝定界程序
// 定义一个最多可容纳1000个活节点的最大堆
MaxHeap<CliqueNode> H(1000);

// 初始化层1
bbnode *E = 0; // 当前的E-节点为根
```

```

int i = 1,    // E - 节点的层
    cn = 0,    // 完备子图的大小
    bestn = 0; // 目前最大完备子图的大小

// 搜索子集空间树
while (i != n+1) { // 不是叶子
    // 在当前完备子图中检查顶点 i 是否与其它顶点相连
    bool OK = true;
    bbnode *B = E;
    for (int j = i - 1; j > 0; B = B->parent, j--)
        if (B->LChild && a[i][j] == NoEdge) {
            OK = false;
            break;
        }

    if (OK) { // 左孩子可行
        if (cn + 1 > bestn) bestn = cn + 1;
        AddCliqueNode(H, cn+1, cn+n-i+1, i+1, E, true);
    }
    if (cn + n - i >= bestn)
        // 右孩子有希望
        AddCliqueNode(H, cn, cn+n-i, i+1, E, false);

    // 取下一个 E - 节点
    CliqueNode N;
    H.DeleteMax(N); // 不能为空
    E = N.ptr;
    cn = N.cn;
    i = N.level;
}

// 沿着从 E 到根的路径构造 bestx[]
for (int j = n; j > 0; j--) {
    bestx[j] = E->LChild;
    E = E->parent;
}

return bestn;
}

```

#### 17.2.4 旅行商问题

旅行商问题的介绍见 16.2.4 节，它的解空间是一个排列树。与在子集树中进行最大收益和最小耗费分枝定界搜索类似，该问题有两种实现的方法。第一种是只使用一个优先队列，队列中的每个元素中都包含到达根的路径。另一种是保留一个部分解空间树和一个优先队列，优先队列中的元素并不包含到达根的路径。本节只实现前一种方法。

由于我们要寻找的是最小耗费的旅行路径，因此可以使用最小耗费分枝定界法。在实现过程中，使用一个最小优先队列来记录活节点，队列中每个节点的类型为 `MinHeapNode`。每个节点包括如下区域： $x$ （从 1 到  $n$  的整数排列，其中  $x[0]=1$ ）， $s$ （一个整数，使得从排列树的根



节点到当前节点的路径定义了旅行路径的前缀  $x[0:s]$ , 而剩余待访问的节点是  $x[s+1:n-1]$ ),  $cc$  (旅行路径前缀, 即解空间树中从根节点到当前节点的耗费),  $lcost$  (该节点子树中任意叶节点中的最小耗费),  $rcost$  (从顶点  $x[s:n-1]$  出发的所有边的最小耗费之和)。当类型为  $MinHeapNode(T)$  的数据被转换为类型  $T$  时, 其结果即为  $lcost$  的值。分枝定界算法的代码见程序 17-9。

程序 17-9 首先生成一个容量为 1000 的最小堆, 用来表示活节点的最小优先队列。活节点按其  $lcost$  值从最小堆中取出。接下来, 计算有向图中从每个顶点出发的边中耗费最小的边所具有的耗费  $MinOut$ 。如果某些顶点没有出边, 则有向图中没有旅行路径, 搜索终止。如果所有的顶点都有出边, 则可以启动最小耗费分枝定界搜索。根的孩子 (图 16-5 的节点 B) 作为第一个 E-节点, 在此节点上, 所生成的旅行路径前缀只有一个顶点 1, 因此  $s=0$ ,  $x[0]=1$ ,  $x[1:n-1]$  是剩余的顶点 (即顶点 2, 3, ..., n)。旅行路径前缀 1 的开销为 0, 即  $cc=0$ , 并且,  $rcost = \min_{i=1}^n MinOut[i]$ 。在程序中,  $bestc$  给出了当前能找到的最少的耗费值。初始时, 由于没有找到任何旅行路径, 因此  $bestc$  的值被设为  $NoEdge$ 。

程序 17-9 旅行商问题的最小耗费分枝定界算法

```
template<class T>
T AdjacencyWDigraph<T>::BBTSP(int v[])
{// 旅行商问题的最小耗费分枝定界算法
    // 定义一个最多可容纳 1000 个活节点的最小堆
    MinHeap<MinHeapNode<T>> H(1000);

    T *MinOut = new T [n+1];
    // 计算 MinOut[i] = 离开顶点 i 的最小耗费边的耗费
    T MinSum = 0; // 离开顶点 i 的最小耗费边的数目
    for (int i = 1; i <= n; i++) {
        T Min = NoEdge;
        for (int j = 1; j <= n; j++)
            if (a[i][j] != NoEdge &&
                (a[i][j] < Min || Min == NoEdge))
                Min = a[i][j];
        if (Min == NoEdge) return NoEdge; // 此路不通
        MinOut[i] = Min;
        MinSum += Min;
    }

    // 把 E-节点初始化为树根
    MinHeapNode<T> E;
    E.x = new int [n];
    for (i = 0; i < n; i++)
        E.x[i] = i + 1;
    E.s = 0; // 局部旅行路径为 x[1:0]
    E.cc = 0; // 其耗费为 0
    E.rcost = MinSum;
    T bestc = NoEdge; // 目前没有找到旅行路径
```

```

// 搜索排列树
while (E.s < n - 1) { // 不是叶子
    if (E.s == n - 2) { // 叶子的父节点
        // 通过添加两条边来完成旅行
        // 检查新的旅行路径是不是更好
        if (a[E.x[n-2]][E.x[n-1]] != NoEdge && a[E.x[n-1]][1] != NoEdge && (E.cc +
            a[E.x[n-2]][E.x[n-1]] + a[E.x[n-1]][1] < bestc || bestc == NoEdge)) {
            // 找到更优的旅行路径
            bestc = E.cc + a[E.x[n-2]][E.x[n-1]] + a[E.x[n-1]][1];
            E.cc = bestc;
            E.lcost = bestc;
            E.s++;
            H.Insert(E);
        }
        else delete [] E.x;
    }
    else { // 产生孩子
        for (int i = E.s + 1; i < n; i++)
            if (a[E.x[E.s]][E.x[i]] != NoEdge) {
                // 可行的孩子, 限定了路径的耗费
                T cc = E.cc + a[E.x[E.s]][E.x[i]];
                T rcost = E.rcost - MinOut[E.x[E.s]];
                T b = cc + rcost; // 下限
                if (b < bestc || bestc == NoEdge) {
                    // 子树可能有更好的叶子
                    // 把根保存到最大堆中
                    MinHeapNode<T> N;
                    N.x = new int [n];
                    for (int j = 0; j < n; j++)
                        N.x[j] = E.x[j];
                    N.x[E.s+1] = E.x[i];
                    N.x[i] = E.x[E.s+1];
                    N.cc = cc;
                    N.s = E.s + 1;
                    N.lcost = b;
                    N.rcost = rcost;
                    H.Insert(N);
                }
            } // 结束可行的孩子
        delete [] E.x; // 对本节点的处理结束

        try {H.DeleteMin(E);} // 取下一个E-节点
        catch (OutOfBounds) {break;} // 没有未处理的节点
    }
}

if (bestc == NoEdge) return NoEdge; // 没有旅行路径
// 将最优路径复制到 v[1:n] 中
for (i = 0; i < n; i++)
    v[i+1] = E.x[i];

while (true) { // 释放最小堆中的所有节点

```

```

delete [] E.x;
try {H.DeleteMin(E);}
catch (OutOfBounds) {break;}
}

return bestc;
}

```

while 循环不断地展开 E-节点，直到找到一个叶节点。当  $s=n-1$  时即可说明找到了一个叶节点。旅行路径前缀是  $x[0:n-1]$ ，这个前缀中包含了有向图中所有的  $n$  个顶点。因此  $s=n-1$  的活节点即为一个叶节点。由于算法本身的性质，在叶节点上  $lcost$  和  $cc$  恰好等于叶节点对应的旅行路径的耗费。由于所有剩余的活节点的  $lcost$  值都大于等于从最小堆中取出的第一个叶节点的  $lcost$  值，所以它们并不能帮助我们找到更好的叶节点，因此，当某个叶节点成为 E-节点后，搜索过程即终止。

while 循环体被分别按两种情况处理，一种是处理  $s=n-2$  的 E-节点，这时，E-节点是某个单独叶节点的父节点。如果这个叶节点对应的是一个可行的旅行路径，并且此旅行路径的耗费小于当前所能找到的最小耗费，则此叶节点被插入最小堆中，否则叶节点被删除，并开始处理下一个 E-节点。

其余的 E-节点都放在 while 循环的第二种情况中处理。首先，为每个 E-节点生成它的两个子节点，由于每个 E-节点代表着一条可行的路径  $x[0:s]$ ，因此当且仅当  $\langle x[s], x[i] \rangle$  是有向图的边且  $x[i]$  是路径  $x[s+1:n-1]$  上的顶点时，它的子节点可行。对于每个可行的孩子节点，将边  $\langle x[s], x[i] \rangle$  的耗费加上  $E.cc$  即可得到此孩子节点的路径前缀  $(x[0:s], x[i])$  的耗费  $cc$ 。由于每个包含此前缀的旅行路径都必须包含离开每个剩余顶点的出边，因此任何叶节点对应的耗费都不可能小于  $cc$  加上离开各剩余顶点的出边耗费的最小值之和，因而可以把这个下限值作为 E-节点所生成孩子的  $lcost$  值。如果新生成孩子的  $lcost$  值小于目前找到的最优旅行路径的耗费  $bestc$ ，则把新生成的孩子加入活节点队列（即最小堆）中。

如果有向图没有旅行路径，程序 17-9 返回 NoEdge；否则，返回最优旅行路径的耗费，而最优旅行路径的顶点序列存储在数组  $v$  中。

### 17.2.5 电路板排列

电路板排列问题（16.2.5节）的解空间是一棵排列树，可以在此树中进行最小耗费分枝定界搜索来找到一个最小密度的电路板排列。我们使用一个最小优先队列，其中元素的类型为 BoardNode，代表活节点。BoardNode 类型的对象包含如下域： $x$ （电路板的排列）， $s$ （电路板  $x[1:s]$  依次放置在位置 1 到  $s$  上）， $cd$ （电路板排列  $x[1:s]$  的密度，其中包括了到达  $x[s]$  右边的连线）， $now$ （ $now[j]$  是排列  $x[1:s]$  中包含  $j$  的电路板的数目）。当一个 BoardNode 类型的对象转换为整型时，其结果即为对象的  $cd$  值。代码见程序 17-10。

程序 17-10 电路板排列问题的最小耗费分枝定界算法

```

int BBArrangeBoards(int **B, int n, int m, int* &bestx)
{
    // 最小耗费分枝定界算法, m 个插槽, n 块板
    MinHeap<BoardNode> H(1000); // 容纳活节点
    // 初始化第一个 E 节点、total 和 bestd
    BoardNode E;

```

```

E.x = new int [n+1];
E.s = 0; // 局部排列为 E.x[1:s]
E.cd = 0; // E.x[1:s]的密度
E.now = new int [m+1];
int *total = new int [m+1];
// now[i] = x[1:s]中含插槽i的板的数目
// total[i] = 含插槽i的板的总数目
for (int i = 1; i <= m; i++) {
    total[i] = 0;
    E.now[i] = 0;
}
for (i = 1; i <= n; i++) {
    E.x[i] = i; // 排列为 12345...n
    for (int j = 1; j <= m; j++)
        total[j] += B[i][j]; // 含插槽 j 的板
}
int bestd = m + 1; // 目前的最优密度
bestx = 0; // 空指针

do { // 扩展 E 节点
    if (E.s == n - 1) { // 仅有一个孩子
        int ld = 0; // 最后一块板的局部密度
        for (int j = 1; j <= m; j++)
            ld += B[E.x[n]][j];
        if (ld < bestd) { // 更优的排列
            delete [] bestx;
            bestx = E.x;
            bestd = max(ld, E.cd);
        }
        else delete [] E.x;
        delete [] E.now;}

    else { // 生成 E - 节点的孩子
        for (int i = E.s + 1; i <= n; i++) {
            BoardNode N;
            N.now = new int [m+1];
            for (int j = 1; j <= m; j++)
                // 在新板中对 插槽计数
                N.now[j] = E.now[j] + B[E.x[i]][j];
            int ld = 0; // 新板的局部密度
            for (j = 1; j <= m; j++)
                if (N.now[j] > 0 && total[j] != N.now[j]) ld++;
            N.cd = max(ld, E.cd);
            if (N.cd < bestd) { // 可能会引向更好的叶子
                N.x = new int [n+1];
                N.s = E.s + 1;
                for (int j = 1; j <= n; j++)
                    N.x[j] = E.x[j];
                N.x[N.s] = E.x[i];
            }
        }
    }
}

```

```

    N.x[i] = E.x[N.s];
    H.Insert(N);
else delete [] N.now;

delete [] E.x; // 处理完当前E-节点

try {H.DeleteMin(E);} // 下一个E-节点
catch (OutOfBounds) {return bestd;} //没有E-节点
} while (E.cd < bestd);

// 释放最小堆中的所有节点
do {delete [] E.x;
    delete [] E.now;
    try {H.DeleteMin(E);}
    catch (...) {break;}
} while (true);

return bestd;
}

```

程序17-10首先初始化E-节点为排列树的根，此节点中没有任何电路板，因此有  $s=0$ ,  $cd=0$ ,  $now[i]=0$  ( $1 \leq i \leq n$ )， $x$ 是整数1到 $n$ 的任意排列。接着，程序生成一个整型数组  $total$ ，其中  $total[i]$  的值为包含 $i$ 的电路板的数目。目前能找到的最优的电路板排列记录在数组  $bestx$  中，对应的密度存储在  $bestd$  中。程序中使用一个do-while循环来检查每一个E-节点，在每次循环的尾部，将从最小堆中选出具有最小 $cd$ 值的节点作为下一个E-节点。如果某个E-节点的 $cd$ 值大于等于 $bestd$ ，则任何剩余的活节点都不能使我们找到密度小于 $bestd$ 的电路板排列，因此算法终止。

do-while循环分两种情况处理E-节点，第一种是处理  $s=n-1$  时的情况，此种情况下，有  $n-1$  个电路板被放置好，E-节点即解空间树中的某个叶节点的父节点。节点对应的密度会被计算出来，如果需要， $bestd$  和  $bestx$  将被更新。

在第二种情况中，E-节点有两个或更多的孩子。每当一个孩子节点  $N$  生成时，它对应的部分排列( $x[1:s+1]$ )的密度 $N.cd$ 就会被计算出来，如果  $N.cd < bestd$ ，则  $N$  被存放在最小优先队列中；如果  $N.cd \geq bestd$ ，则它的子树中的所有叶节点对应的密度都满足  $density \geq bestd$ ，这就意味着不会有优于  $bestx$  的排列。

## 练习

3. 在程序17-4中增加代码，将指向由函数  $AddLiveNode$  生成的节点的指针存储在一个链表队列中。 $MaxLoading$  必须利用这些指针信息在程序终止之前删除所有生成的节点。

4. 本节所使用的  $AddLiveNode$  函数直到程序终止前才删除所生成的节点。实际上，没有活动孩子且不产生叶节点的那些节点都可以被立即删除。类似地，在第  $n$  层节点中，若节点没有重量为  $bestw$  的孩子，则可以立即删除该节点。讨论怎样尽快删除不需要的节点。描述实现这种方法时所涉及的时间/空间变化。你推荐使用上述方法吗？

5. 在程序17-6中，定义一个  $bestw$  来记录目前生成的可行节点所对应的重量的最大值。修改程序17-6，使得如果活节点的重重量大于等于  $bestw$ ，则将它加入子集树及最大堆中。此外，还必须增加初始化和更新  $bestw$  的代码。

6. 只使用一个最大优先队列, 来实现用最大收益分枝定界方法求解货箱装船问题, 即不要使用程序17-6中所用到的部分解空间树, 而在每个优先队列的节点中都加入通向根节点的路径信息。

7. 修改程序17-6, 把删除bbnode类型和HeapNode类型节点的任务放在程序结尾处。

8. 只使用一个最大优先队列, 利用最大收益分枝定界法求解 0/1 背包问题, 即不必保存一个部分解空间树, 所有优先队列中的节点都记录着通往根节点的路径。

9. 修改程序17-7, 使得删除bbnode和HeapNode类型的节点的任务放在程序的结尾处执行。

10. 1) 程序17-8中, 若右孩子的un值大于等于bestn, 则将它加入最大堆中, 如果将条件设为 $un > bestn$ , 程序能否正确执行呢? 为什么?

2) 程序是否将 $un = bestn$ 的左孩子加入最大堆中?

3) 修改程序, 使得只将 $un > bestn$ 的节点加入到最大堆和生成的解空间树中。

11. 考察最大完备子图问题的解空间树。对于任意层 (第  $i$  层) 的子树中的节点  $x$ , 令  $MinDegree(x)$  为  $x$  所包含的顶点的度的最小值。

1) 证明任何以  $x$  为根的子树的叶节点都不可能表示一个尺寸超过  $X.un = \min\{X.cn + n - i + 1, MinDegree(X) + 1\}$  的完备子图。

2) 使用以上  $X.un$  的定义重写 BBMaxClique。

3) 比较两种 BBMaxClique 版本在运行时间及产生解空间树节点的数目上的不同。

12. 只使用最大优先队列, 实现最大完备子图问题的最大收益分枝定界算法。即: 不必保存一个部分解空间树, 而在每一个最大优先队的节点内包含通向根的路径。

13. 修改程序17-8, 使得删除bbnode和CliqueNode类型的节点的工作放在程序结尾处执行。

14. 修改程序17-9, 使得  $s = n - 2$  的节点不进入优先队列, 并且, 将当前最优排列放在数组 bestp 中。当下一个 E-节点的  $lcost \geq bestc$  时, 算法终止。

15. 使用指向父节点的指针来实现部分解空间树, 并使用包含  $lcost$ ,  $cc$ ,  $rcost$  和  $ptr$  (指向解空间树中对应节点的指针) 域的优先队列来实现程序 17-9。

16. 写出用 FIFO 分枝定界方法求解电路板排列问题的代码。代码必须输出最优电路板排列的排列次序及对应的密度。使用合适的数据来测试代码的正确性。

17. 用 FIFO 分枝定界方法来搜索一种电路板的排列, 使得最长的网组的长度最小 (参见 16 章练习 17)。

18. 使用最小耗费分枝定界法来完成练习 17。

19. 用最小耗费分枝定界算法求解 16 章练习 18 的顶点覆盖问题。

20. 用最大收益分枝定界算法求解 16 章练习 19 的简易最大切割问题。

21. 用最小耗费分枝定界算法求解 16 章练习 20 的机器设计问题。

22. 用最小耗费分枝定界算法求解 16 章练习 21 的网络设计问题。

23. 用 FIFO 分枝定界算法求解 16 章练习 22 的  $n$ -皇后放置问题。

\* 24. 用 FIFO 分枝定界完成 16 章练习 23。

\* 25. 用 FIFO 分枝定界完成 16 章练习 24。

\* 26. 用 FIFO 分枝定界完成 16 章练习 25。

\* 27. 用最小耗费分枝定界完成 16 章练习 23。

\* 28. 用最小耗费分枝定界完成 16 章练习 24。

\* 29. 用最小耗费分枝定界完成 16 章练习 25。

\* 30. 用任意的分枝定界方法完成 16 章的练习 25。在本练习中, 必须把增加活节点的函数以及选择下一个 E-节点的函数作为函数的参数。